**Table of Contents**

[**CHAPTER 1: INTRODUCTION TO APACHE KAFKA 2**](#_30j0zll)

[Theory 2](#_1fob9te)

[**An Overview of BigData 2**](#_2et92p0)

[Quick Introduction to Hadoop 2](#_tyjcwt)

[*Why Hadoop? 2*](#_3dy6vkm)

[Quick Introduction to Hadoop Distributed File System 3](#_1t3h5sf)

[*Block Placement in HDFS 4*](#_4d34og8)

[*HDFS Architecture 5*](#_2s8eyo1)

[Quick Introduction to Spark 7](#_17dp8vu)

[*What is Spark? 7*](#_3rdcrjn)

[*Why Spark? 7*](#_lnxbz9)

[*Components of Spark 9*](#_z337ya)

[*Spark Data Storage 12*](#_2xcytpi)

[Introduction to Kafka 12](#_1ci93xb)

[*What is Kafka? 12*](#_3whwml4)

[*Why Kafka? 13*](#_3as4poj)

[Confluent Overview 16](#_49x2ik5)

[Kafka Use Case 17](#_147n2zr)

[Introduction to ZooKeeper 19](#_3o7alnk)

[*What is ZooKeeper? 19*](#_23ckvvd)

[*ZooKeeper Data Consistency 20*](#_32hioqz)

[*ZooKeeper Architecture 21*](#_1hmsyys)

[*Why does kafka need ZooKeeper? 22*](#_41mghml)

[**LAB EXERCISE 24**](#_2grqrue)

[**SUMMARY 25**](#_vx1227)

[**REFERENCES 26**](#_34g0dwd)

# CHAPTER 1: INTRODUCTION TO APACHE KAFKA

## Theory

This chapter is intended to provide a comprehensive introduction to Apache Kafka, which is the center of focus throughout this book. We shall go through a brief overview of BigData before we begin with Kafka.

## An Overview of BigData

### Quick Introduction to Hadoop

Apache Hadoop is an open source distributed framework that allows storage and processing of large data (BigData) sets across cluster of commodity machines. Hadoop overcomes the traditional limitations of storing and computing of data by distributing the data over cluster of commodity machines, making it scalable and cost-effective.

The idea of Hadoop was originated when Google released a white paper about **Google File System (GFS)** - a computing model built by Google which was designed to provide efficient, reliable access to data using large clusters of commodity hardware. The model was then adopted by Doug Cutting and Mike Cafarella for their search engine called “Nutch”. Hadoop was then developed to support distribution for the Nutch search engine project by Doug Cutting and Mike Cafarella. Well, what does the name Hadoop mean? There is no significance for the name and it is not an acronym either. Hadoop is the name that Doug Cutting’s son gave to his yellow stuffed elephant. The name is very unique, easy to remember and sometimes funny. Not only does Hadoop have such name with no significance but also its sub-projects tend to have such names, which are based on names of animals, like Pig, and for the same reason. They are unique, not used anywhere else and are easy to remember.

### Why Hadoop?

Companies today have been realizing that there is lot of information in unstructured documents spread across the network. A lot of data is available in the form of spreadsheets, text files, e-mails, logs, PDF’s and other data formats that contain valuable information which can help discover new trends, design new products, improve existing products, know customers better and what not. Data is increasing at an alarming rate beyond limits like never before and there are no signs of slowing down, at least in the near future. To deal with such data, we need a reliable and low cost tool to meaningfully process it. Therefore, we use Hadoop. Hadoop helps us process all this BigData which is present in variety of formats reliably, in a very lesser time and in a flexible and cost-effective way.

Let us see why Hadoop is so popular and what it has in store for you.

* **Scalable**: Hadoop is scalable, meaning: you can just start from a single node server and eventually increase more nodes as you need more storage and more computing power.
* **Fault-Tolerant**: Hadoop helps prevent loss of data. All the data which is stored in Hadoop Distributed File System is broken into blocks and stored with a default replication factor of 3. While processing data, if a node goes off, the process does not stop but continues, as the data still exists in other nodes.
* **Flexible**: Hadoop does not require schema. Hadoop can process unstructured, semi-structured and structured data from any kind of source or even from multiple sources.
* **Cost-effective**: Hadoop does not require expensive high end computing hardware. Hadoop works well with a cluster of commodity machines by parallel computing.

### Quick Introduction to Hadoop Distributed File System

**Hadoop Distributed File System (HDFS)** is a File System which extends over a cluster of commodity machines rather than a single high end machine. HDFS is a distributed large scale storage component and is highly scalable. HDFS can accept node failures without losing data. HDFS is widely known for its reliability. Let us now check out why HDFS stands out of crowd when it comes to Distributed file systems.

|  |  |
| --- | --- |
| **Reliable Data Storage** | HDFS is very much reliable when it comes to data storage. Whatever the data stored in HDFS is replicated by a default replication factor of 3. That means, even if a machine fails, the data will be still available in two other machines. |
| **Cost Effective** | HDFS can be deployed on cluster of commodity hardware and can save you a lot of bucks. High end expensive hardware is not required by HDFS to function. |
| **Big Datasets** | HDFS is capable of storing Petabytes of data over a cluster of machines where a file can range from Gigabytes to Terabytes of size. HDFS is not designed to store huge number of small sized files as the file system meta data is stored in memory of NameNode. |
| **Streaming Data Access** | HDFS provides streaming access to data. HDFS is best suited for batch processing of data and not suitable for interactive processing. HDFS is not designed for applications which require low latency access to data such as OnLine Transaction Processing (OLTP). |
| **Simple Coherency Model** | HDFS is designed to *write once and read many times* access model for files. Appending the content to files is supported at the end but cannot be updated at arbitrary point, and it is also not possible to have multiple writers. Files can only be written by a single writer. |

### Block Placement in HDFS

Hadoop is designed in such a way that the first block replica is placed on the same node as of client, and the second replica is placed on a different rack from first replica. Third replica is placed on a random node on the same rack as of the second replica. If the replication factor is more, random nodes in the cluster are selected to place the replicas. If a client running outside the cluster stores a file, random node (That isn’t busy) is picked to place the first replica. This way, if a node fails, the data is still available on other nodes of the cluster, and if a rack fails, again, the data is still intact.

### HDFS Architecture

HDFS is a Master and Slave architecture, in which the Master node controls and assigns jobs to all its slave nodes. The following terminologies are used to describe the Master and Slave nodes.

The Master Nodes in HDFS are:

* NameNode
* Secondary NameNode

The Slave Nodes in HDFS are:

* Data Nodes

These nodes are the core serving roles in HDFS architecture. Let us now look in detail about the roles of each Node and understand them better.

|  |  |
| --- | --- |
| **NameNode** | NameNode is a HDFS daemon which controls all the Data Nodes and handles all the File System operations such as creating a directory, creating a file or reading and writing a file. The NameNode is responsible for managing the File System namespace image. It holds the image in memory, representing how the File System looks like. It also maintains the meta data for all the blocks of files in the File System and also tracks the replication value, so it knows the locations of blocks stored on Data Nodes within the cluster. But the meta data is not stored onto the disk and is every time recreated when it starts. NameNode stores all this information persistently on local disk in the form of namespace image and edit log. The NameNode is the *single point of failure* in the Hadoop cluster. If the NameNode fails, entire cluster fails. |
| **Data Nodes** | Data Nodes are the slave machines controlled by the NameNode, that actually does all the block operations. Data Nodes store and retrieve blocks when asked to do so by NameNode, and periodically inform NameNode with the lists of blocks they store by sending heartbeats. Data Nodes replicate the data physically when instructed by the NameNode on where and how to replicate. |
| **Secondary NameNode** | Secondary NameNode, as its name implies, is not exactly the Secondary NameNode. The secondary NameNode is not a high availability solution and does not automatically take over the responsibilities of NameNode on failure. Its main role is to create checkpoint and take the backup of NameNode periodically. It is like a backup solution to the NameNode. The hardware specifications of secondary NameNode should be similar to that of NameNode. In case of NameNode’s failure, the secondary NameNode can be manually configured to work as a primary NameNode. This is not a high availability solution. |

Now that we have had a quick introduction to Hadoop, let us shift our focus on the main topic of our discussion, Apache Spark.

## Quick Introduction to Spark

### What is Spark?

Apache Spark is an open source, fast and unified parallel large-scale data processing engine. It provides a framework for programming with distributed processing of large datasets at high speed. Spark supports the most popular programming languages such as Java, Python, Scala and R. Spark is scalable, meaning, it can run on a single desktop machine or a laptop to a cluster of thousands of machines. Spark provides a set of inbuilt libraries which can be accessed to perform data analysis over a large dataset. However, if your requirement doesn’t get satisfied with the inbuilt libraries, you can write one or explore countless external libraries from open source communities on the internet.

### Why Spark?

Why use Spark when we have Hadoop? Well, Spark excels as a unified platform for processing huge data at very high speeds for various data processing requirements (will be discussed later in this chapter). Also, Spark is an in-memory processing framework. Spark is considered as a successor of Apache Hadoop. Let us briefly discuss the advantages of Spark over Hadoop.

With the Hadoop ecosystem, we had various frameworks for various data processing requirements. As a developer, you would use MapReduce framework to analyze your data using any of the languages such as Java, C++, Python etc, but a data warehouse engineer who is a SQL expert, has to learn any of the aforementioned programming languages. To overcome this problem, a new framework which runs on the top of Hadoop called “Hive” was introduced. This was also a problem for ETL processing, and so “Pig” was introduced. Similarly tools like “Giraph” and “Mahout” were introduced for Graphs processing and Machine Learning respectively. Moreover, Hadoop is only used for batch processing and there was no way to process data in real time. So, for this a new framework called “Storm” was integrated with Hadoop to work with streaming data. With so many frameworks, organizations had a tough time to maintain all the frameworks and track issues pertaining to them. Fortunately, all this would change with the advent of Spark. As mentioned, Spark is a unified platform which provides all these frameworks as one package with four major components.

Now, what actually does In-memory processing mean? Aren’t all the applications processed in memory only? Well, yes, all the applications are processed in-memory and written back to disk when processing is done, but Spark can process data in-memory and also retain the data within the memory or write to disk. Let us understand this with a figure by comparing Spark with MapReduce.

***1(a) Data Processing with MapReduce***





*Read Write* *Read* *Write* *Read Write Read Write*



In MapReduce, the data present in HDFS or any other *Distributed file system* is read by a MapReduce application and is processed in memory and then written back to disk after the job is complete. If the processed data is again needed for further processing, the data is again read from disk by a MapReduce application, processed in memory and then written back to disk. This process continues as per the requirement, as seen in the figure 1(a). The processes of reading and writing data from and to the disk increase the IO latency and so the overall job duration is increased. This is optimized in Spark as shown in the figure 1(b).

***1(b) Data Processing with Spark***







*Read*



*Write*

In Spark, the data is read from the disk, processed in-memory but, instead of spilling it back to disk, Spark can retain the data within the memory for further processing. So, if the processed data is again required for further processing, the data is already present in the memory and the Spark application processes the data eliminating the IO latency, and therefore the overall time to process the job is significantly reduced. With this, the processing speed when compared to MapReduce has been increased up to 100 times faster. The processed data from a Spark application can either be retained in memory or can be stored to the disk as per the requirement, as shown in the figure 1(b).

The reasons, such as a unified platform for various data processing requirements and High Speed In-Memory processing, have gained worldwide popularity throughout the industry with almost all the major organizations using Spark for their data processing requirements.

### Components of Spark

Now that we know why Spark is being used, let us dive in more and learn what Spark is made up of. Let us look at each of the major Spark’s components individually and know them in detail. The following figure 1(c) shows the components of Spark.

***1(c) Components of Spark***









Let us look at a brief explanation regarding these components so that we can better understand the Spark components.

|  |  |
| --- | --- |
| **Spark Core** | Spark Core, as the name suggests, is the core component of Spark which has all the basic functionalities for processing large datasets. Some of its functionalities include managing memory, scheduling jobs, fault tolerance, using in-memory computation, referring datasets stored in storage systems etc. Spark Core includes a programming abstraction (API) called Resilient Distributed Datasets also known as RDDs, which is responsible for partitioning data across nodes on a cluster. With the help of these RDDs, the data can be transformed, collected and reduce things together. These RDD APIs can be referred by using any of the programming languages such as Scala, Python, Java and R as shown in the figure 1(c). |
| **Spark SQL** | The Spark SQL component provides the developer with an SQL like interface to work with huge structured data which is distributed over a cluster of nodes. Spark SQL works well with structured and semi structured data. Spark SQL can also work with data sources such as Apache Hive tables, Avro, JDBC, ORC, JSON and Parquet file formats. Spark SQL also allows developers to combine RDD APIs along with Spark SQL code in a single application. |
| **Spark Streaming** | Spark Streaming component of spark deals with processing of real time data also known as Streaming data. The streaming data can be from fleet of web servers, sensors, IOT devices or any other sources which generate data. This enables Spark to ingest data as it is generated in realtime and perform data manipulation on that data. There are three major phases of Spark Streaming. They are *Gathering*, *Processing* and *Data Storage*. Spark Streaming is also fault tolerant and scalable. We will talk a little about Spark Streaming in this book. |
| **Spark MLlib** | Spark MLlib is short for Machine Learning libraries which provides Machine Learning for huge datasets. MLlib contains various Machine Learning algorithms such as *Regression*, *Clustering*, *Classification* and *Collaborative Filtering*. MLlib also contains lower level primitives such as generic gradient descent optimization algorithm. MLlib also uses the linear algebra package called *Breeze* for numerical computing. |
| **GraphX** | GraphX deals with processing of Graphs in very efficient and distributed manner. GraphX extends the RDD APIs, which allows a developer to create a directed multigraph with properties attached to each vertex and edge. |
| **Cluster Managers** | Spark is all about processing massive amounts of datasets by distributing them over a number of nodes and scaling the cluster as required. In order to efficiently perform this task, a cluster manager is required, and Spark has its own cluster manager called *Standalone Scheduler*. Spark can also be deployed using *Hadoop YARN,* *Apache Mesos* or *Kubernetes* as a cluster manager to schedule jobs and manage the resources of the cluster. |

### Spark Data Storage

Spark supports major file systems such as HDFS, Amazon S3, Azure Blob etc. Spark supports the local file system for storing the data as well. However, using a distributed file system such as HDFS can leverage the power of Spark by distributing the datasets throughout the cluster. Spark is also capable of dealing with various file formats such as text, ORC, parquet etc.

Hadoop and Spark are used to analyze huge amounts of data. This only solves one of the challenges faced with BigData. The other challenge is to actually collect huge amounts of data efficiently. Kafka helps us with this challenge. Let us now proceed with an introduction to Kafka and see how it deals with this challenge.

## Introduction to Kafka

### What is Kafka?

Kafka is an open-source, distributed, persistent and fault-tolerant message streaming platform or a central repository, which can handle high volume (trillions) of Publish-Subscribe messages every day. Publish-Subscribe messaging system is a system where data is produced (Publish) by producers and consumers consume (subscribe) the data. We shall be looking at Producers and Consumers in detail, in the next chapter.

Kafka is written in Scala and is built on top of the ZooKeeper coordination service. The integration of Spark and Kafka enables real-time streaming data analysis. Kafka was built at LinkedIn and later donated to the Apache Software Foundation, making it open-source.

Kafka is popular because of the following features:

* **Scalable**: Kafka can be scaled from a single machine to a cluster of machines spanning data centers with zero downtime. The number of machines required can be scaled as per the requirement.
* **Persistent**: The data or messages are stored and cached in disk instead of memory, making them persistent and durable. Kafka is also fault-tolerant with replications and partitions.
* **Performance:** Kafka provides great performance and stability with huge volumes of publishing and subscribing messages.
* **Distributed**: Kafka is distributed to a cluster of machines and hence processes streams with great speed and efficiency.
* **Real-Time Streaming:** Kafka is capable of processing streams of messages in real-time as they occur.

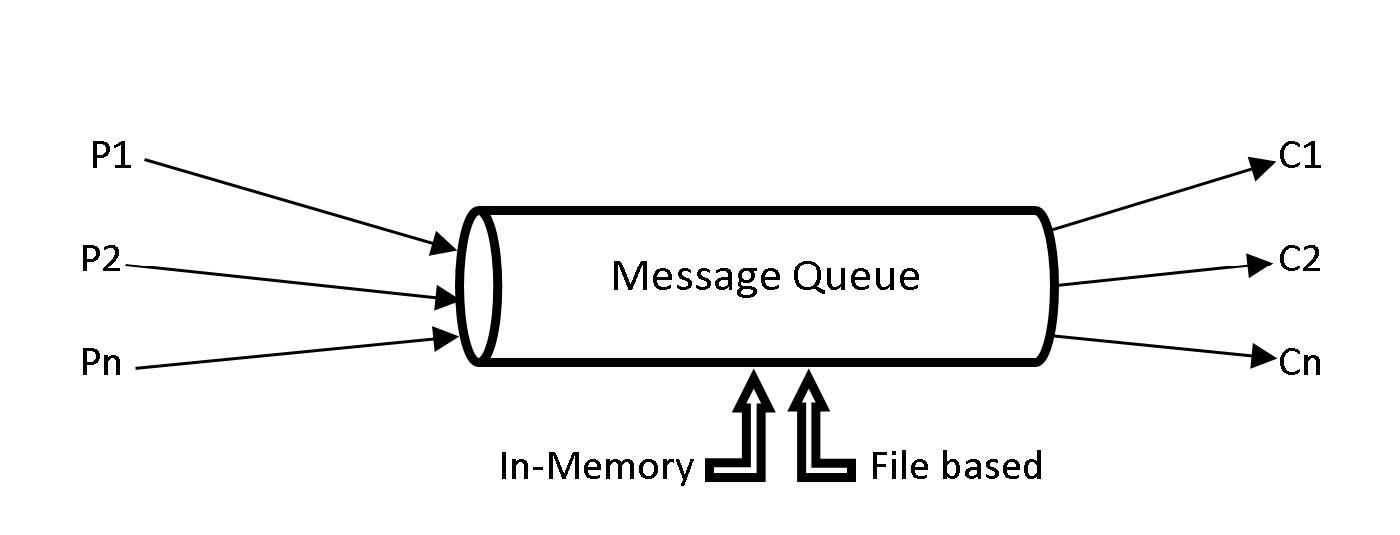
Kafka is used to build real-time streaming data pipelines for steady transfer of data between applications. The use cases include collecting logs from multiple servers, data from sensors etc.

### Why Kafka?

Why do we need Kafka? Don’t we already have message queue or data streaming platforms? How is Kafka better than the traditional data streaming platforms? Let us answer all these questions now. To understand why we need Kafka, we should first understand how the traditional streaming platforms work.

Consider a traditional system with a message queue as shown below. The message queue could be implemented in any of the programming languages. The message queue receives messages from various numbers of processes. These messages are denoted by P1, P2…Pn. This message queue may store data in-memory or on the file system based on the implementation. If the system is memory based, the data will be lost in case of system failure. But if the system is file based, the data will be intact even if the system goes down. The data from the message queue will be consumed by various numbers of consumer processes. These processes are denoted by C1, C2…Cn.

The data is produced by the producers and the consumers consume the data via the message queue. So far so good. However, the problem arises when one of the consumers is connected to a distributed platform such as Spark application. Spark is capable of processing huge amounts of data in a distributed manner. The message queue is not distributed and is implemented in a single machine. Therefore, the traditional message queue is limited by resources of that machine such as CPU core, RAM and disk size, and becomes the bottleneck as it cannot receive huge data similar to Spark.



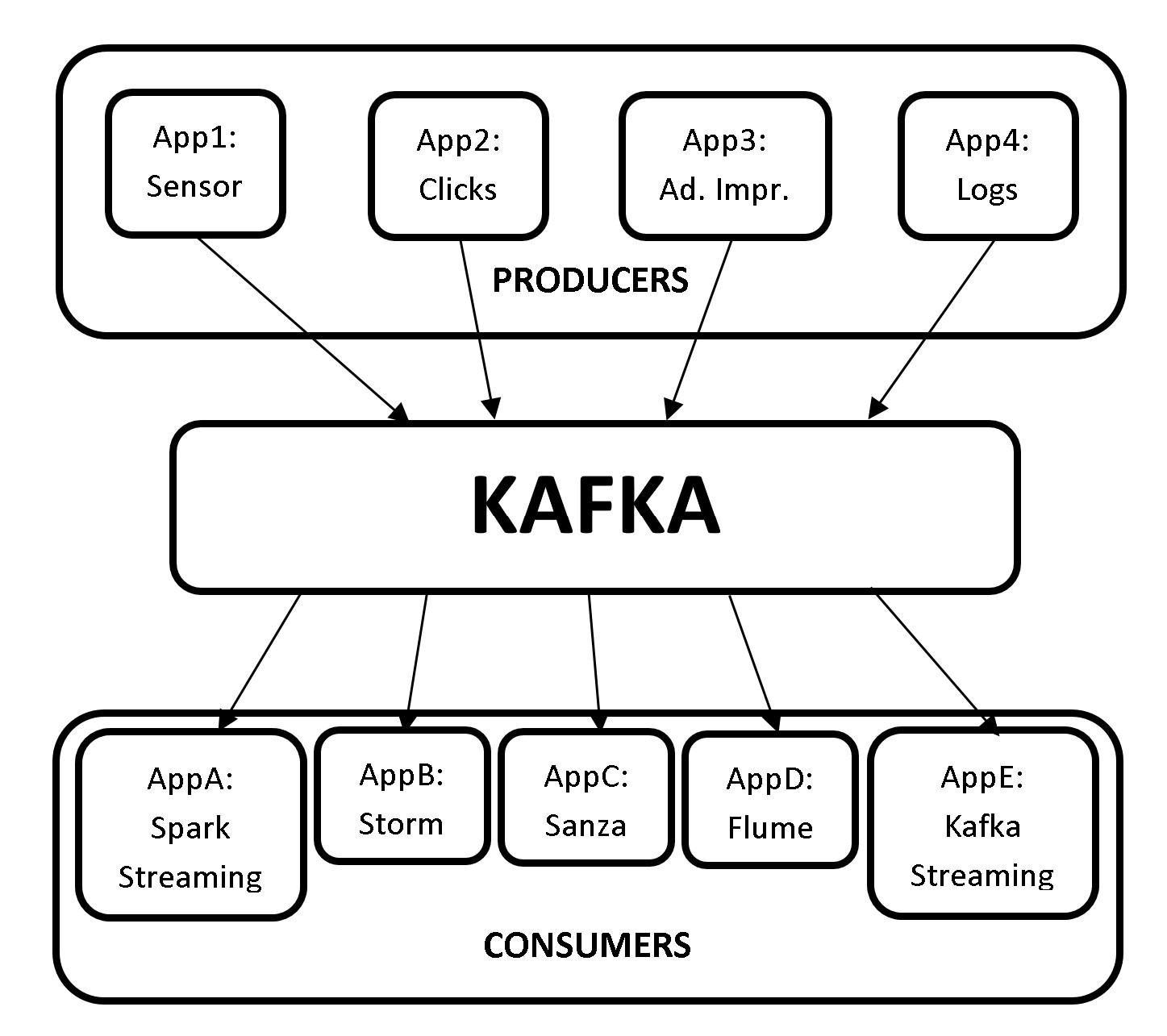
***1(d) Traditional Message Queue***

In order to cater a distributed processing application such as Spark, it is only efficient to have a message queue distributed across several machines. The limitation of machine resources such as CPU, RAM and disk was one of the reasons to develop Kafka, which is a distributed message queue.

Let us now look at another reason why Kafka had to be developed. Consider an application *app1* which generates data during its operations. This data is required by an analytics application, say *appA*. The data is simply transferred via an interface from *app1* to *appA*. Later there comes another application, say *app2* that generates data. This data is also required by the analytics application *appA* for more analytics. So, an interface has to be implemented to send the data from *app2* to *appA*. Once the interface for *app2* is implemented and tested, the interface for *app1* should also be tested in order to make sure the new implementation has not broken something with the old implementation.

Eventually more operational (*app3, app4, app5…*) and analytics (*appB, appC, appD…*) applications were developed which require data from operational to analytics applications. For example, data from *app4* might be needed by *appB* and *appD* or data from *app3* might be required by *appA* and *appC.* There might be many such possibilities for transferring data from one or more operational applications to one or more analytics applications. Every time a new operational application generates data for analytics application, a new interface has to be implemented, and all the interfaces should be tested to check if something is broken due to the new implementations. This becomes very hard to maintain when there are so many applications and interfaces. The entire system should be tested every time a new interface between applications is implemented.

All these problems lead to the development of Kafka. Instead of having different interfaces for different applications, all the operational applications send the data to Kafka. The analytics applications can then consume the data from Kafka, making it a central repository. The figure below shows a pictorial representation of how data is being produced and consumed with Kafka as a central repository.



***1(e) Kafka Message Queue***

As seen from the picture above, we need not build interfaces and test them to transfer data every time a new application is implemented. All the producers generate the data to Kafka and the consumers pull the data from Kafka, making it a central repository.

## Confluent Overview

Confluent is a data streaming platform based on Apache Kafka, and is founded by the creators of Apache Kafka. Confluent expands the capabilities of Apache Kafka not only to Publish-Subscribe messages, but also to a full-scale event streaming platform that enables to store and process real-time streams. Confluent Data Streaming Platform consists of Apache Kafka as core component.

Confluent data streaming platform provides the following components, making it a complete distribution of Apache Kafka.

* **Apache Kafka:** Apache Kafka is the core component of Confluent Platform. Apache Kafka is an open-source, distributed, persistent and fault-tolerant message streaming platform or a central repository, which can handle high volume (trillions) of Publish-Subscribe messages every day.

However, Apache Kafka is not a complete data streaming platform. It only provides data storage and interfaces for reading and writing data. It does not directly integrate with other services such as RDBMS. With Confluent’s other components, the capabilities of Kafka can be extended such that it can integrate with other services.

* **Kafka Connect:** Kafka Connect is used to transfer data to and from Kafka. HDFS, JDBC, S3, Elasticsearch, etc. are some of Kafka connectors that transfer data to and from Kafka.
* **Kafka REST Proxy:** The Kafka REST proxy provides a RESTful interface to a Kafka cluster. The Kafka REST proxy can be used to send and receive messages, view the state of the cluster and perform administrative actions.
* **Kafka Streams:** Kafka Streams is a powerful yet easy to use client library for stream processing and analysis. With Kafka Streams processing layer, we can perform transformations or analysis by reading the real-time data and write the results back to Kafka.
* **Schema Registry:** Schema Registry is a serving layer for metadata. Schema Registry provides a RESTful interface for storing and retrieving AVRO schemas. It makes sure that the data which is being sent and received is in a common format i.e., checking schema compatibility for Kafka. We shall look at this in detail in the upcoming chapters.
* **KSQL:** KSQL is a streaming SQL engine for Kafka to run queries on data stored in Kafka cluster. KSQL is used internally on Kafka streams for processing.

We shall be only be focusing at Apache Kafka throughout this book. However, let us look at a use case to better understand how all these components are integrated to form an end-to-end pipeline using Confluent Data Streaming platform.

## Kafka Use Case

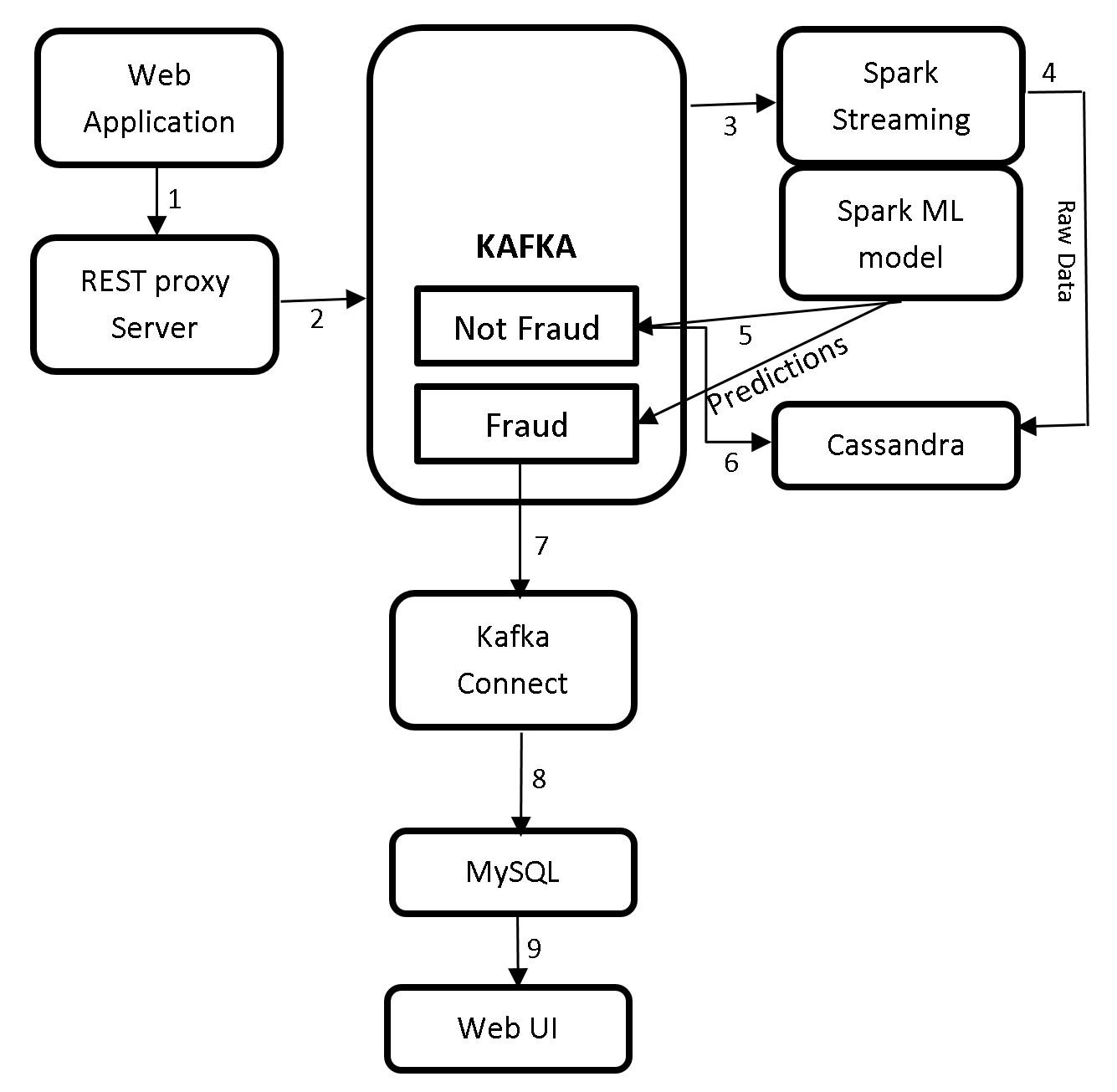
Let us now look at a Fraud Detection use case at high level, to better understand how all the components of Confluent can be integrated to form an end-to-end pipeline. We shall be looking at an use case of fraud detection in credit card transactions.

The usage of credit cards has been on the rise from the past few decades, and so are the frauds revolving around cards. People make transactions using credit cards round the clock. This transaction data can be analysed to prevent fraud in real-time and minimize monetary loss.

1. Let’s consider that all the credit card transactions are recorded in a web application.
2. These transactions are then transferred to Kafka message queue using the REST proxy server. The REST proxy server provides an interface to Kafka.
3. The transactions are then pulled from Kafka by Kafka Streaming or Spark Streaming to apply transformations or analysis. *Please note that Spark Streaming is generally used to process real-time messages similar to credit card transactions, since we feed the processed data to a Machine Learning model.*

After processing the streams, the data is internally fed to a Spark Machine Learning model to predict if the transaction is a fraud or not. Confluent Kafka does not provide Machine Learning module and hence Spark MLlib will be used for predictions.

1. The unpredicted raw data from Spark Streaming is also saved to Cassandra for further processing, if required.



***1(f) Kafka Use Case***

1. There will be two outcomes for this predictions i.e., genuine transactions and potential fraud transactions. This data is sent to Kafka.
2. Usually there will be more genuine or non-fraud transactions than the fraud transactions. Hence, all the genuine transactions are sent to a NoSQL database such as Cassandra.
3. The potential fraud transactions are then transferred to a RDBMS database such as MySQL using Kafka Connect.
4. These records can then be pulled from MySQL and displayed on webUI.
5. The domain experts can then take necessary actions to determine if these predicted potential fraud transactions are actually fraud. If it is determined that a transaction is fraud, the card issuer can block or hold the card from making transactions to prevent further loss.

This is how the Confluent Kafka components can be used to build an end-to-end pipeline.

Before we wrap up this chapter let us see what is ZooKeeper and why does Kafka need ZooKeeper.

## Introduction to ZooKeeper

### What is ZooKeeper?

ZooKeeper is an open source, robust distributed coordination service for distributed applications. ZooKeeper is an open source Apache Software Foundation project, it’s available for free and ready to use. ZooKeeper helps in overcoming many of the common challenges faced by distributed applications. ZooKeeper can be used for synchronization, sequential consistency and coordination between distributed applications. It helps in maintaining the configuration information which can be shared to all the nodes in a distributed system. ZooKeeper also helps in group services such as leader election and many more. ZooKeeper is reliable and fast yet very simple to work with. With ZooKeeper you can build reliable, distributed data structures for group membership, leader election, coordinated workflow, and configuration services. You can also build generalized distributed data structures like locks, queues, barriers, and latches.

ZooKeeper provides an eventually consistent view of its *znodes* which are nothing but files or directories in a file system. It provides basic CRUD operations such as creating, updating, and deleting *znodes*. It provides an event-driven model in which clients can watch for changes to specific znodes, for example if a new child is added to an existing *znode*. ZooKeeper is a high availability service as it consists of a set of ZooKeeper servers known as *Ensemble* (cluster), with each of the server holding an in-memory image of the distributed file system to serve client read requests. Each server also holds a persistent copy on disk.

One of the servers in the *Ensemble* is dynamically chosen by consensus as the leader, and all other servers are followers. The leader is responsible for all writes and for updating the changes to its followers. When the majority of followers update a change successfully, the write succeeds and the data is still available even if the leader fails. When a leader fails, a new leader is again dynamically chosen by consensus within the *Ensemble*. This eliminates the single point of failure scenario, and the *Ensemble* keeps on doing its work as it should.

When a client connects to ZooKeeper, it is provided with the list of servers in the *Ensemble*. The client connects to one of the servers in the ensemble at random until a connection is established. Once connected, ZooKeeper creates a session with a pre-specified timeout period by client. The ZooKeeper client automatically sends heartbeats periodically to keep the session alive if no operations are performed for a while, and automatically handles failover. If the connection between ZooKeeper and client fails, the client automatically detects this and retries to connect to a different server in the *Ensemble*. After it is reconnected, the same client session is retained while the failure has occurred.

### ZooKeeper Data Consistency

ZooKeeper provides with the following guaranteed consistencies:

**Sequential consistency:** Updates from a client to the ZooKeeper service are applied in the order they are sent. Since all writes go through the leader, the global order is simply the order in which the leader receives write requests.

**Single System Image:** The Single System Image guarantees that a client will see the same view of the ZooKeeper service regardless of the server in the ensemble that it is connected to.

**Atomicity:** There are no partial failures. The updates from a client to ZooKeeper service either succeed or fail. For example, assume a client sends an update to a server, but before the response is received the network connection is lost or the server goes down. Now, did the update get through to the server? If yes, did the operation complete successfully? The only way to know the answers to all these questions is when the server/network is back up again. ZooKeeper though cannot help with network problems or partial failures, but it handles through atomicity. If the network/server goes down during an update operation, the operation is marked failed, else it is marked as success.

**Reliability:** If the update is successful, it is persistent and will not be rolled back. The update will only be overwritten when client makes a new update. The updates are still available even when the server fails.

**Consistent Client View:** A client’s view of the system is guaranteed to be up-to-date within a certain time bound, generally within tens of seconds. If a client does not observe system changes within that time bound, then the client assumes a service outage and will connect to a different server in the *Ensemble*.

### ZooKeeper Architecture

The ZooKeeper architecture consists of leader and follower servers. The collection of these servers is known as *Ensemble.* The number of servers in a ZooKeeper Ensemble should always be an odd number. The reason behind this is because we need majority during the voting process of electing a leader. Let us now look at the responsibilities of leader and follower servers.

* **Leader**: When an *Ensemble* is first started, voting process to choose a leader takes place. During the voting process a leader is elected and the process is complete as soon as a simple majority of followers have synchronized their state with the leader. After leader election is complete, leader is responsible to handle all the write requests from clients, and changes are committed to all followers. Once a majority of followers have persisted the change, the leader commits the change and notifies the client of a successful update. There should always be a leader. If leader is down, all the existing followers go through the voting and elect a new leader.
* **Followers**: Followers function is similar to that of the leader by allowing clients to connect them and send, read, and write requests to them, but the writes are forwarded to leader.
* **Observers:** Observers are the non-voting members of an *Ensemble* which do not participate in the voting process, but only hear the voting results*.* When there are more followers who participate in voting, the write performance significantly drops, and hence Observers are added to *Ensemble* as to address this issue. Observers improve ZooKeeper’s scalability, and we can increase the number of Observers as much as we like without harming the performance of votes. Observers function is exactly the same as Followers, where clients connect to them and send, read, and write requests to them. Observers forward these requests to the Leader like Followers do, but they then simply wait to hear the result of the vote.

## Why does Kafka need ZooKeeper?

Kafka cannot be started without ZooKeeper. We must first start ZooKeeper service before we start Kafka. Based on the explanation of Zookeeper in the sections above, Kafka needs ZooKeeper for the following.

* **Electing a Controller:** Kafka consists of brokers to handle requests such as sending and receiving messages. The broker acts as a mediator for both producers and consumers to handle the requests. A broker is a Kafka server. Multiple brokers form a Kafka cluster. Since a Kafka cluster has multiple brokers, we need to elect a leader among these brokers to maintain the cluster state. The broker which we elect as a leader is called controller and is responsible to maintain the leader-follower relationships. In case of a failure of a broker, the controller’s responsibility is to tell all the replicas to act as partition leaders, in order to fulfill the duties of the partition leaders on the broker that is about to fail. ZooKeeper is used to elect this controller. ZooKeeper also makes sure there is only one leader, and also elects a new leader in case of failure.
* **High Availability:** ZooKeeper keeps tracks of membership of all the brokers, and periodically checks if any of the brokers that are part of the cluster have failed. It maintains High availability of Controller Broker.
* **Topic Configuration:** ZooKeeper makes track of existing topics, partitions for each topic, replica locations, preferred leader and configuration override information for each topic.
* **Access Control Lists:** ZooKeeper maintains Access Control Lists for each topic, i.e., the read and write permissions of clients.

Do not worry if you do not understand the new concepts at this time. We shall look at brokers, replications, partitions, topics etc. in detail in the next chapter. It will be clear why we need ZooKeeper once we understand the architecture of Kafka in detail.

That’s all for the theory for this chapter.

# LAB EXERCISE

"There are no activities required for this lab"

# SUMMARY

Kafka is an open-source, distributed, persistent and fault-tolerant message streaming platform or a central repository, which can handle high volume (trillions) of Publish-Subscribe messages every day. Publish-Subscribe messaging system is a system where data is produced (Publish) by producers and consumers consume (subscribe) the data.

Kafka is written in Scala and is built on top of the ZooKeeper coordination service. The integration of Spark and Kafka enables real-time streaming data analysis. Kafka was built at LinkedIn and later donated to the Apache Software Foundation, making it open-source.

ZooKeeper is an open source, robust distributed coordination service for distributed applications. ZooKeeper is an open source Apache Software Foundation project, is available for free and ready to use. ZooKeeper helps in overcoming many of the common challenges faced by distributed applications. ZooKeeper can be used for synchronization, sequential consistency and coordination between distributed applications. It helps in maintaining the configuration information which can be shared to all the nodes in a distributed system.

# THE KAFKA FRAMEWORK

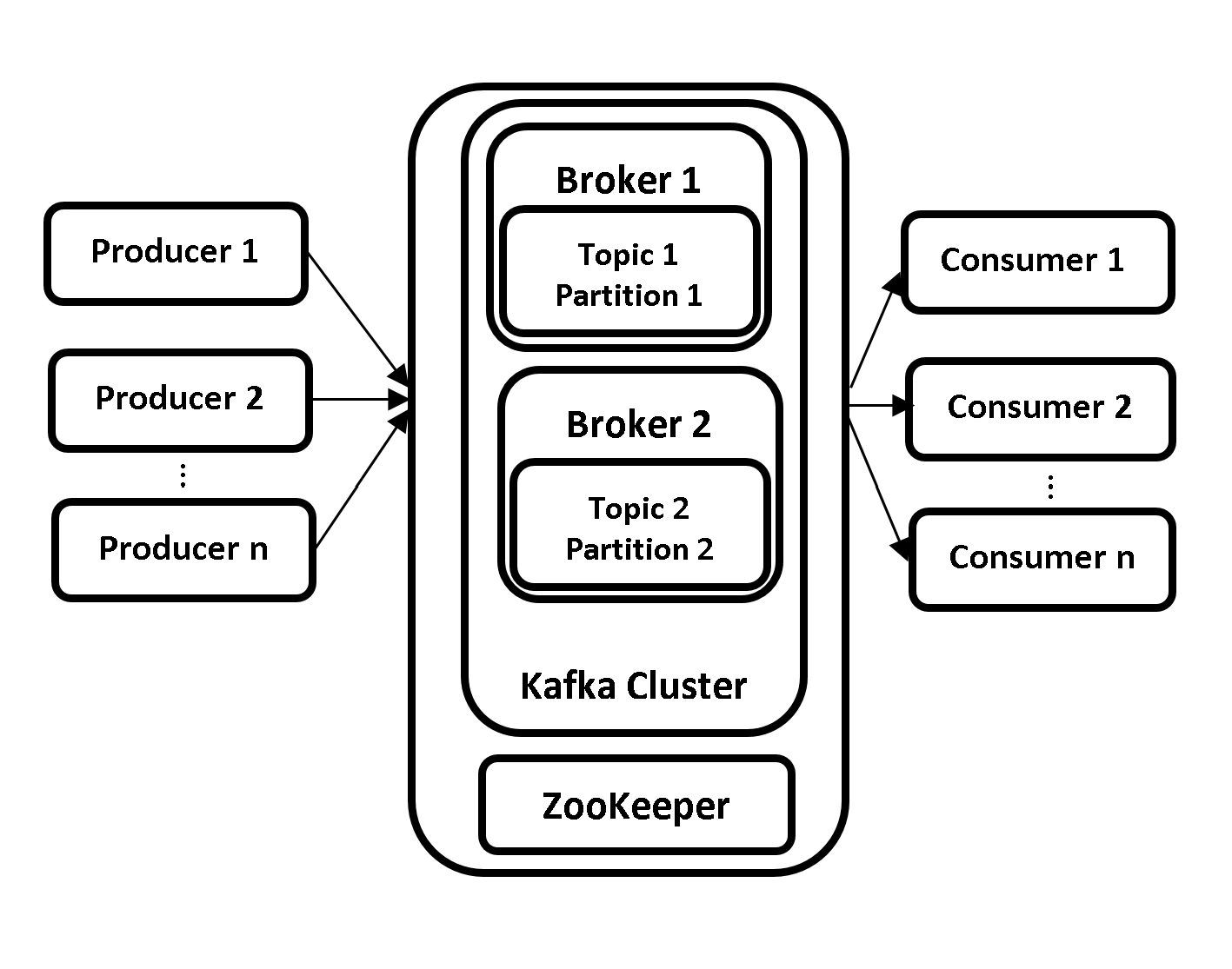
## Theory

In the previous chapter, we have looked at an overview of BigData with brief introduction to Apache Hadoop and Apache Spark frameworks. We have also had a comprehensive introduction of Apache Kafka and Apache ZooKeeper. Let us now deep dive and learn more about Apache Kafka. Let us first look at the architecture, APIs and CLI of Apache Kafka and then proceed to the labs to and install Apache ZooKeeper and Apache Kafka on to our machines.

## Kafka Architecture

We have looked at an use case of Kafka in the previous chapter, which shows a high level representation of Kafka architecture. Let us now look at the Hive architecture in detail.

The Kafka architecture is as shown in the picture below.



***2(a) Kafka Architecture***

The Kafka architecture consists of the following components.

* Brokers
* Producers
* Consumers
* Topics
* Partitions
* Replications
* ZooKeeper

Let us now look at overview of each of the component.

### **Topics**

Kafka is capable of streaming messages from multiple sources. A topic is a name given to each source in order to distinguish and store the incoming messages. Topics are used to group similar messages together. Producers write messages to Topics and Consumers read data from Topics with the help of Brokers. A new directory is created with the name of the topic’s partition whenever a new topic is created. A leader broker is responsible for all the read and write operations for that partition.

### **Partitions**

Kafka Topics are further divided into Partitions to achieve scalability. A Topic can have multiple Partitions. A Partition is a log where the messages are appended as received. The messages are distributed across multiple Brokers with the help of Partitions, thus making Kafka a distributed message queue. This allows multiple Consumers to read data from a topic in parallel.

### **Producers**

Producers are the Kafka clients which create and send new messages to the Brokers. Producers create messages for one or more than one Kafka Topics. Producers are the source of data for Kafka Cluster. There can be multiple producers in a Kafka cluster. Each message produced contains an offset. Offset is the integer metadata associated with each message which increase for each message that is produced by the Producer.

### **Consumers**

Consumers are the Kafka clients which read data from Brokers. There can be multiple consumers in a Kafka cluster. Consumers read the data from Topics to which they are subscribed to. The data is read in the order of which it was produced. The Consumers know which messages have already been consumed with the help of Offsets. Partitions have a unique offset for each message. Consumer simply stores the last offset it consumed in ZooKeeper or Kafka. This will help Consumer to restart consuming the messages exactly from where it stopped.

### **Brokers**

Kafka cluster consists of daemons known as Brokers. A Kafka cluster consists of one or more Brokers. Brokers are the most important components and work horses of a Kafka Cluster and, without Brokers there is no Kafka. As the name suggests, Brokers act as middlemen between the Producers and Consumers. Brokers receive messages from Producers, assign them with offsets and store them on disk into Kafka Topics. Similarly, it responds to Consumers’ data pull requests from Kafka Topics.

In the production environment, there should be only one Broker per node in the Kafka cluster. However, there can be multiple brokers on a single node in the testing environment. The Brokers are further classified as Leader Broker and Controller Broker.

#### Controller Broker

A Controller Broker is one of the Broker that is available in the pool of Brokers in a Kafka cluster. A Controller Broker is elected with the help of ZooKeeper as soon as the Brokers in Kafka cluster are started and, is considered as Master Broker. A Controller Broker is similar to any other Broker in the Kafka Custer but with extra responsibilities. There is only one active Controller Broker at all times.

The Controller Broker is responsible for assigning partitions to brokers, monitoring Brokers for failures and rebalancing partitions to other brokers in the event of failure. The Controller Broker is also responsible for the duties like any other Broker in the cluster, i.e., leading partitions, performing reads/writes and having partition replications.

#### Leader Broker

Each Partition in a Kafka cluster has one Broker which acts as a Leader Broker. If the partition is created with a replication factor of more than one, the partitions are replicated to the Follower Brokers. Leader Broker manages the read/write requests for the partitions from Producers and Consumers and, the Follower Brokers simply replicate the partitions in Leader Broker. In case of failure of a Leader Broker, another Broker having the replications will take over the leadership making it redundant.

### **Replications**

The partitions are replicated across the Kafka cluster to achieve high availability. This is to ensure that the data is not lost in case of a Broker failure in Kafka cluster. The replications are of two types. Leader and Follower Replicas. The messages transmitted from leader broker are leader replicas and the follower replicas are for fault tolerance in case of data loss available in follower brokers.

### **ZooKeeper**

As learned in the previous chapter, ZooKeeper provides coordination service for Kafka cluster. Please check Why does Kafka need ZooKeeper? for more information.

We shall look at internals of all these concepts in detail in the upcoming chapters.

That’s all for the theory for this chapter. Let us move to the lab exercise to install ZooKeeper and Kafka into our machines.

# AIM

The aim of the following lab exercises is to install and configure ZooKeeper and Kafka.

The labs for this chapter include the following exercises.

* Downloading and Installing JDK
* Downloading and Installing ZooKeeper
* Configuring ZooKeeper
* Downloading and Installing Kafka
* Configuring Kafka
* Starting ZooKeeper and Kafka

We need the following packages to perform the lab exercise:

* Java Development Kit
* Apache ZooKeeper
* Apache Kafka

We will be using Ubuntu 18.04 LTS operating system with at least 4GB of RAM throughout the book for all our exercises. Please make sure you install this version of OS before proceeding with Kafka installation.

# LAB EXERCISE 1: KAFKA INSTALLATION

1. **Download and install JDK**
2. **Download and install ZooKeeper**
3. **Configure ZooKeeper**
4. **Download and install Kafka**
5. **Configure Kafka**
6. **Start ZooKeeper and Kafka**

## Task 1: Download and Install JDK

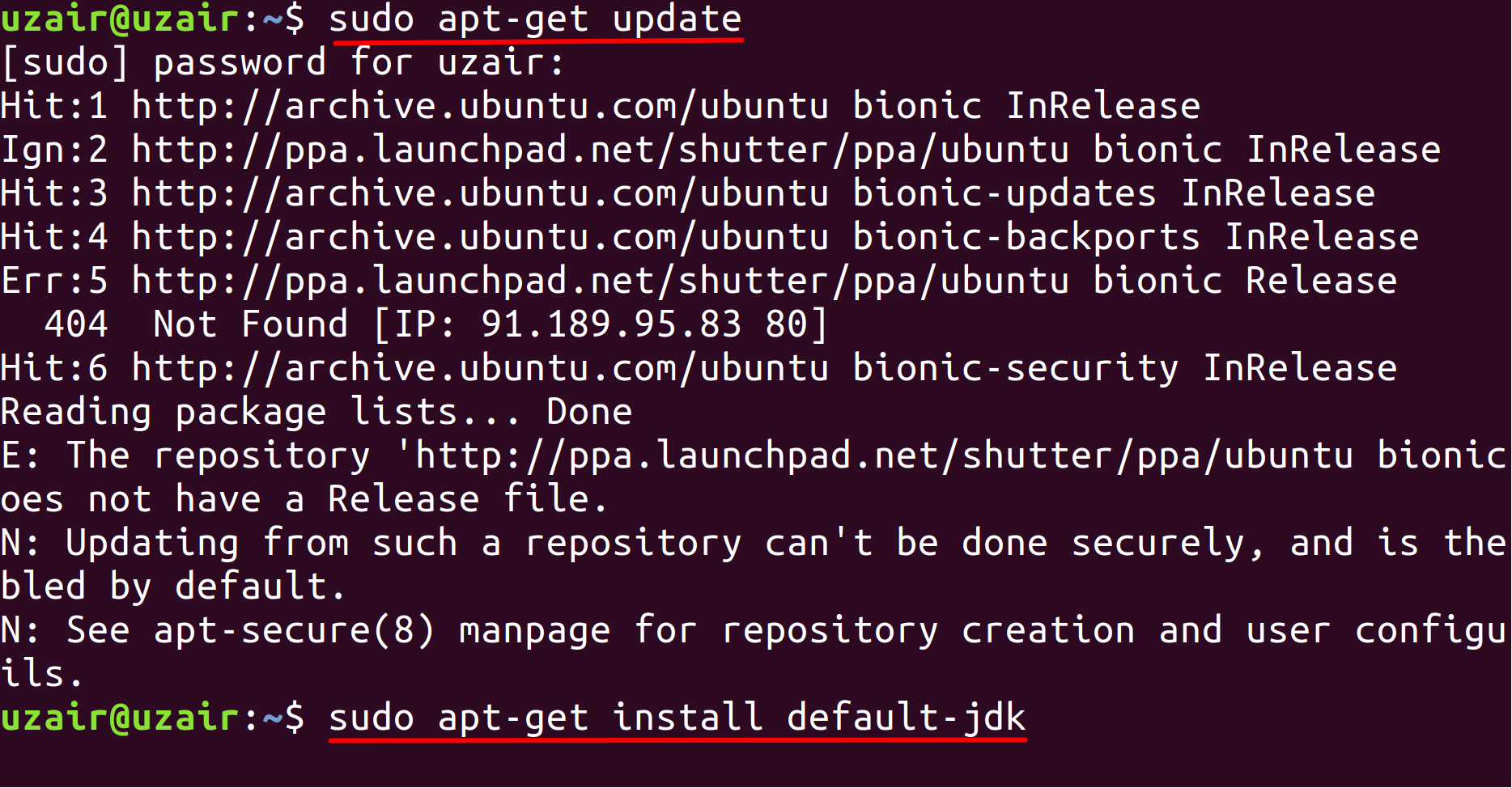
**Step 1:** From the terminal, run the following commands to install JDK (Java Development Kit).

$ sudo apt-get update

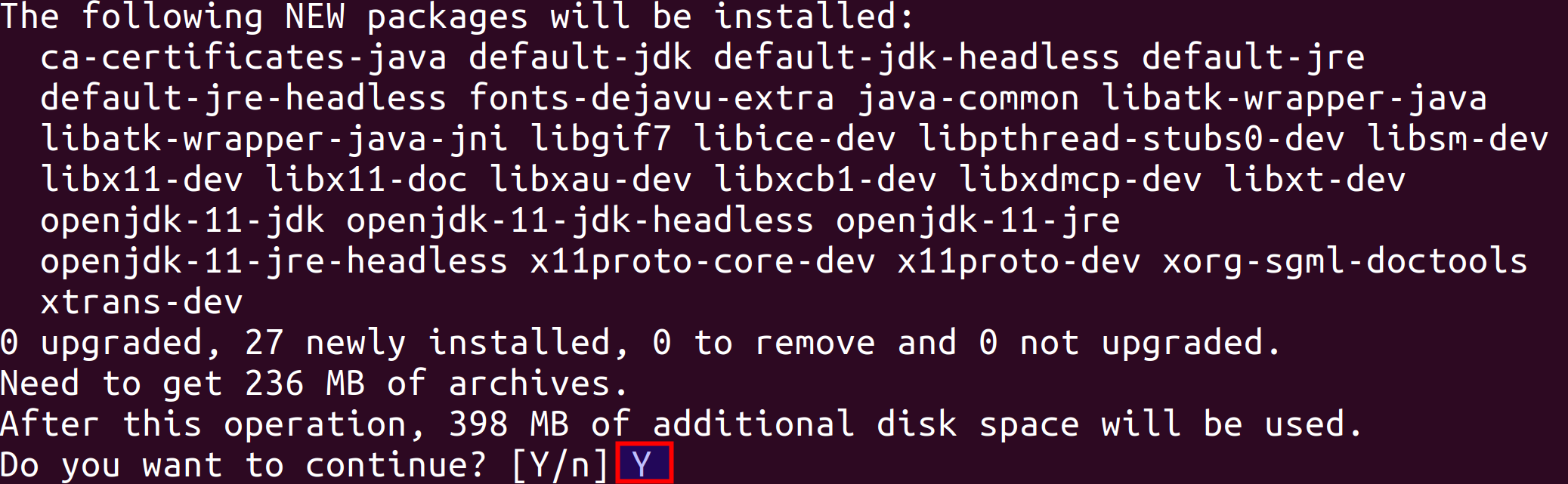
This will update the package index. You might be asked to enter your password after you run the command above.

**Step 2:** Once you run the above command, run the following command to actually download and install JDK.

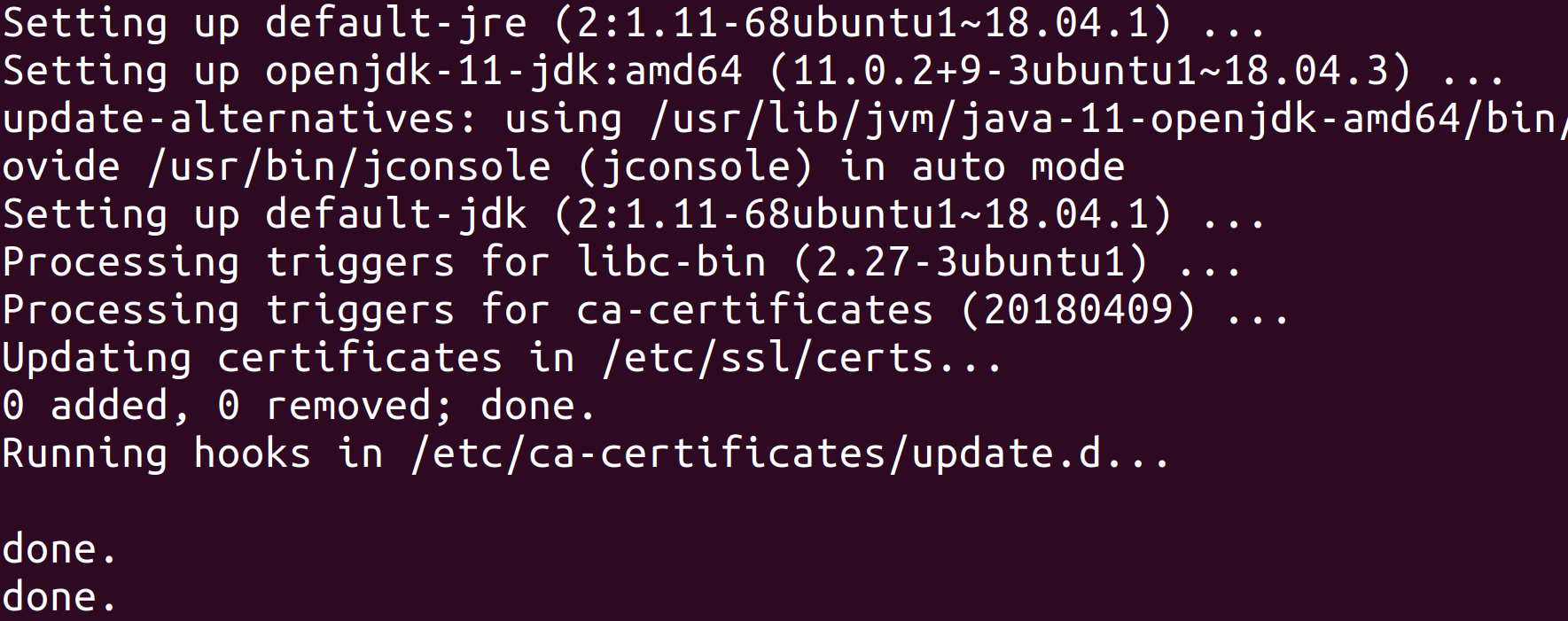
$ sudo apt-get install default-jdk



The prompt will ask you to hit ‘Y’ after running the above command as shown in the screenshot. Hit ‘Y’ from your keyboard to continue with the installation and finally hit the Enter key. This will download and install JDK on your machine.

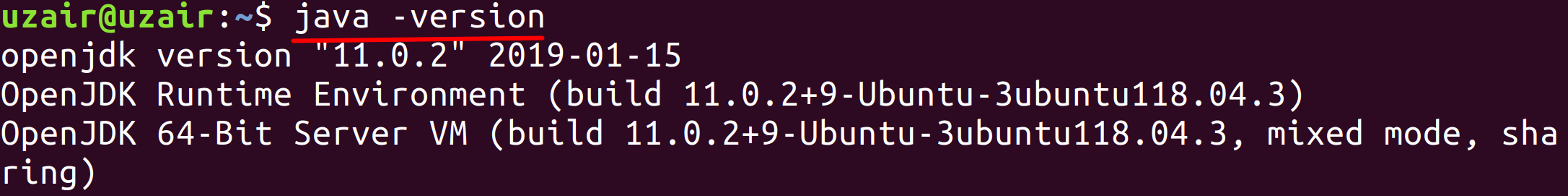


The installation process might take some time depending on your internet connection. Please allow it to download and install completely. You should see the following message when the installation is complete.



**Step 3:** Run the following command to check if Java has been installed successfully. The terminal should show the Java version as shown in the screenshot.

$ java –version



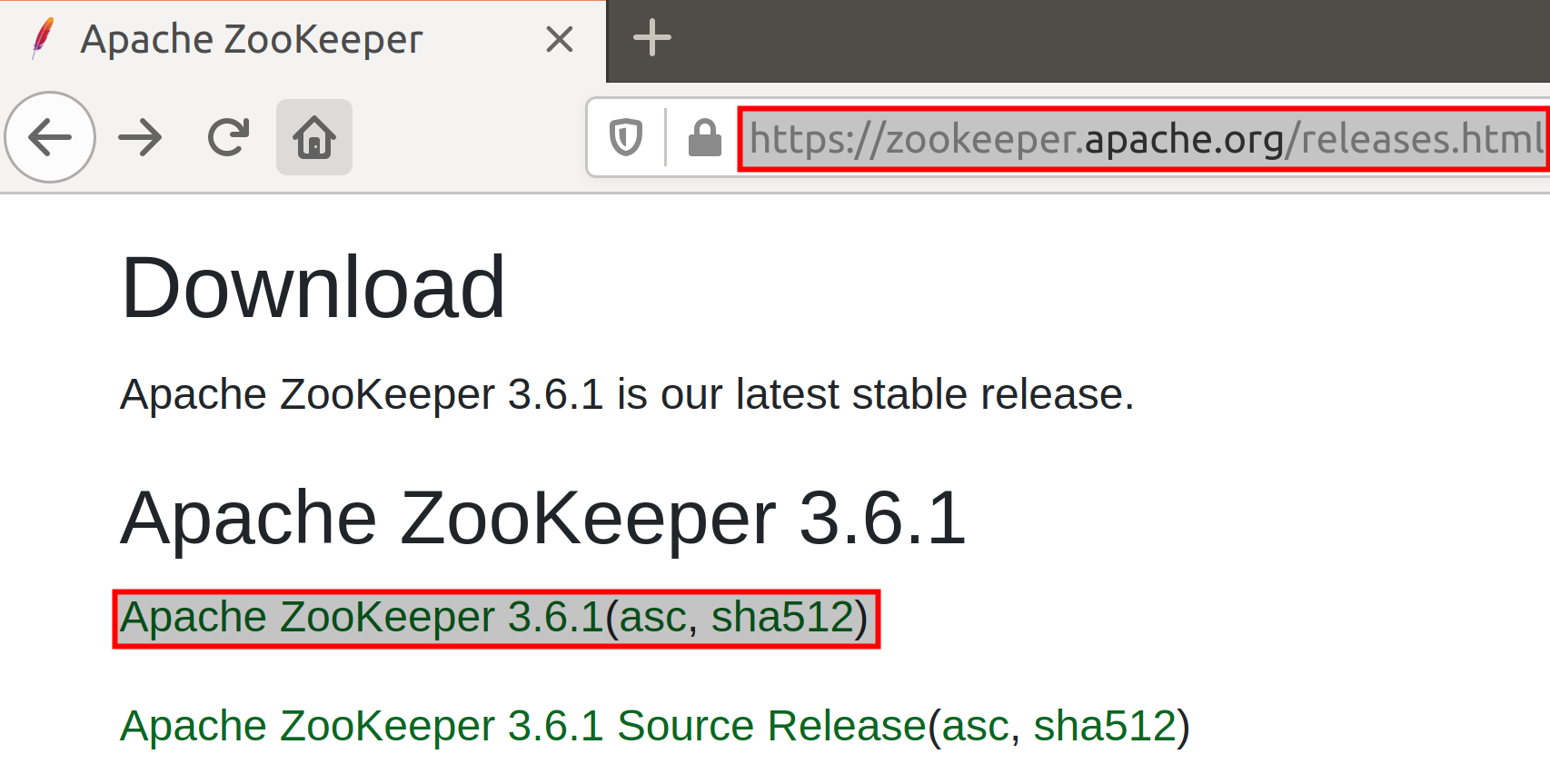
Please note that your version of JDK might be latest version from what you see in the screenshot.

**Task 1 is complete!**

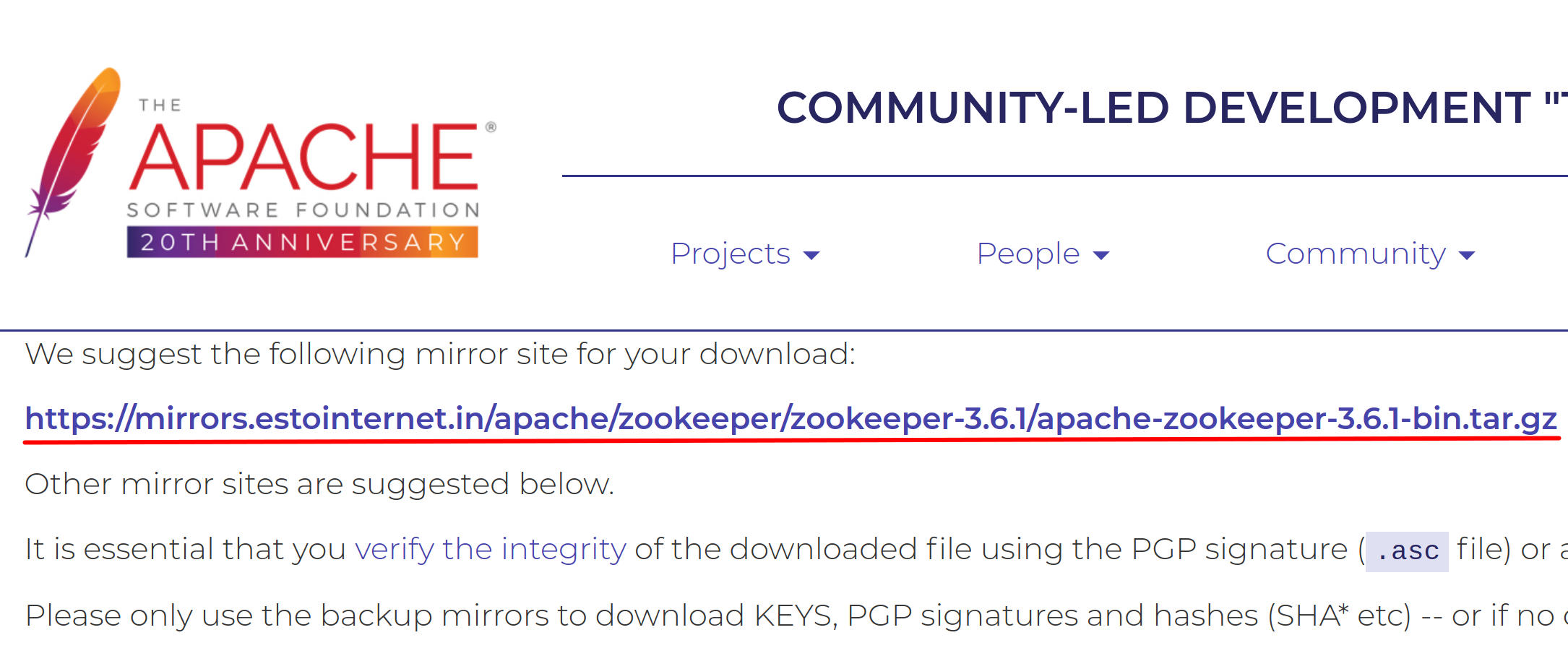
## Task 2: Download and Install ZooKeeper

**Step 1:** Let us ZooKeeper in standalone mode. Navigate to the download URL below and click to download the latest stable version for ZooKeeper (which is ZooKeeper 3.6.1 at the time of writing this book).

Download URL : <https://zookeeper.apache.org/releases.html>



After clicking the download link, you will be taken to a page with mirror site to download ZooKeeper. Click the mirror link as shown below and your download should start. The download may take a while depending upon your internet connection.



**Step 2:** The download will be saved to the *Downloads* directory by default. Execute the following command from your terminal to change the directory to *Downloads* folder.

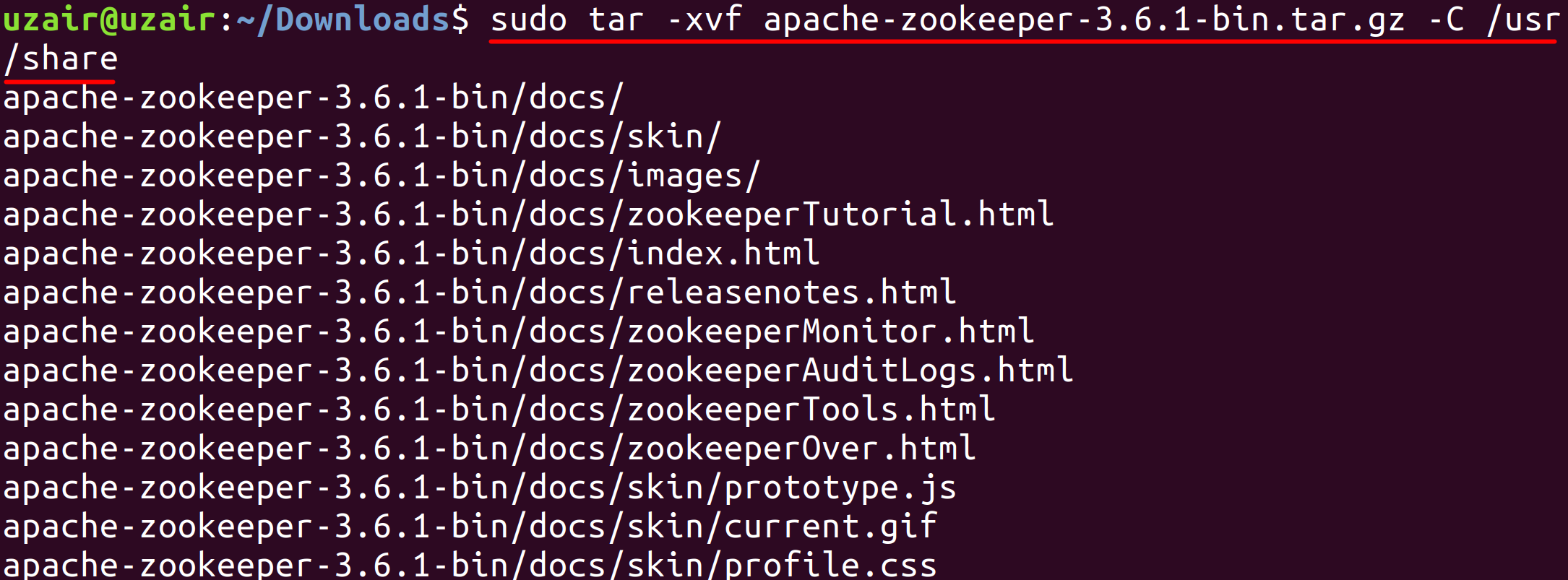
$ cd Downloads

Once you are in the *Downloads* directory, you may optionally check if ZooKeeper has been downloaded using the *ls* command.

$ ls

Now that you are sure that you have the ZooKeeper tar file, untar the ZooKeeper tar file to */usr/share* directory using the command below.

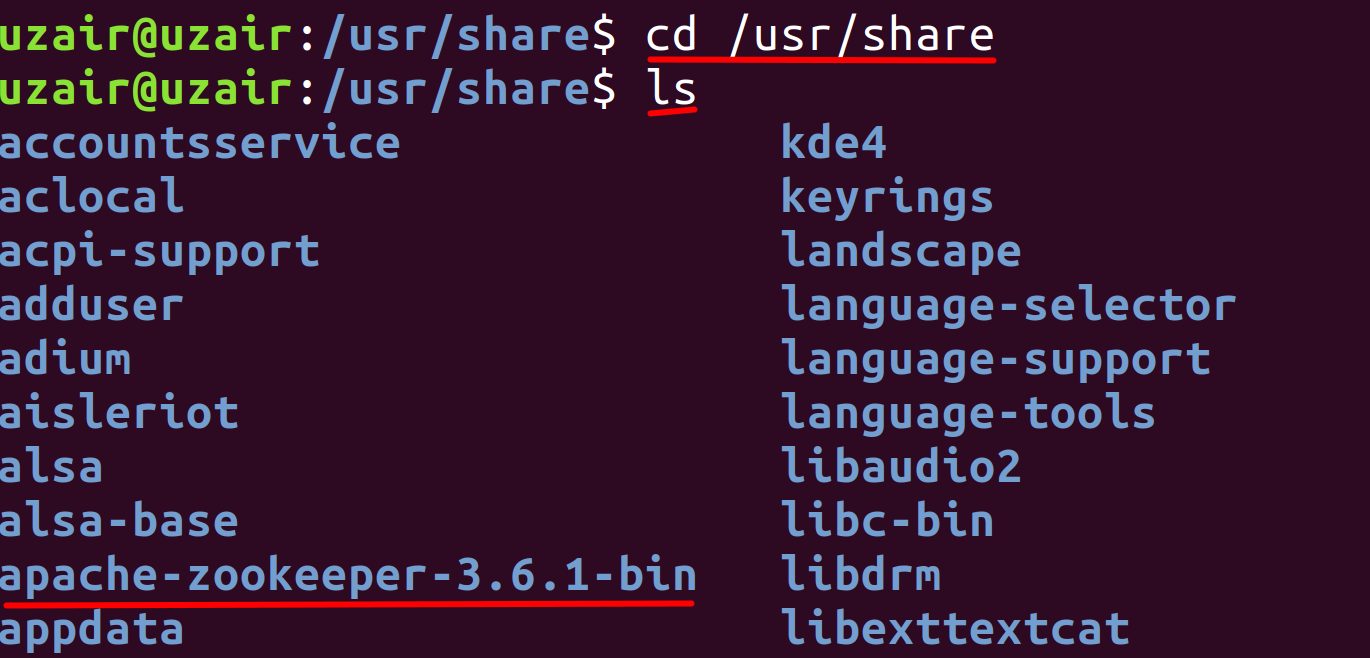
$ sudo tar –xvf apache-zookeeper-3.6.1-bin.tar.gz –C /usr/share

****

The file will start to untar to */usr/share* directory as shown in the screenshot above. You can verify the same by executing the following command below.

$ cd /usr/share

$ ls

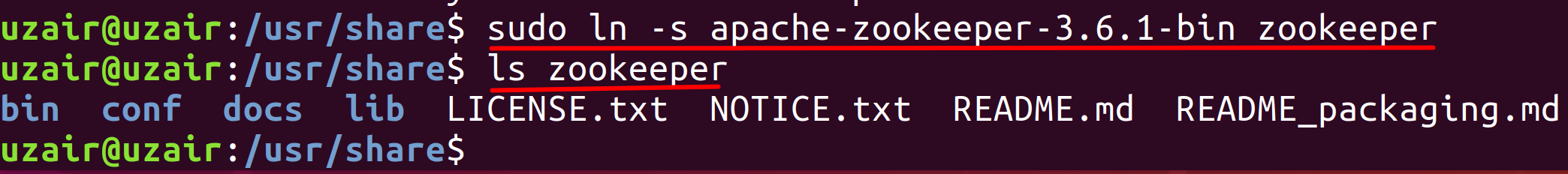


As we can see from the screenshot above, the ZooKeeper directory is listed.

**Step 3:** Let us make a softlink to the ZooKeeper directory so that we don’t have to refer ZooKeeper with entire name as above. This will also be useful for the future updates. Execute the following command.

$ sudo ln –s apache-zookeeper-3.6.1-bin zookeeper

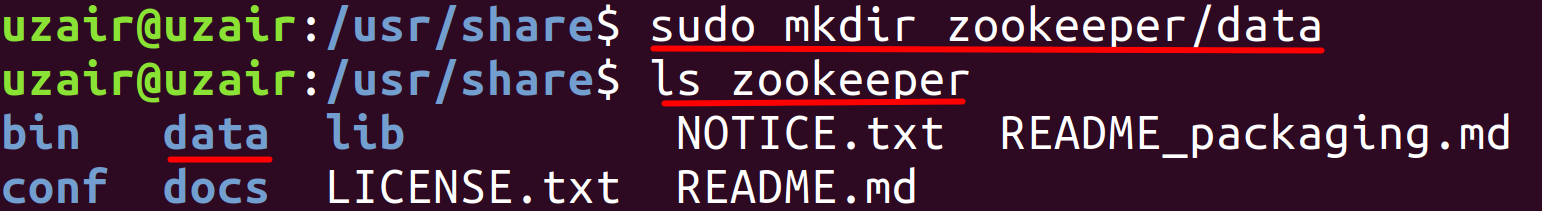
Run the following command again to check if we were able to create the softlink successfully.



**Step 4**: Let us conclude the installation process by creating a directory where ZooKeeper will store its data.

$ sudo mkdir zookeeper/data

$ ls zookeeper



**Step 5:** Change the permissions of the ZooKeeper directory by running the following command:

$ sudo chown <username> /usr/share/zookeeper

<username> - Insert your username.

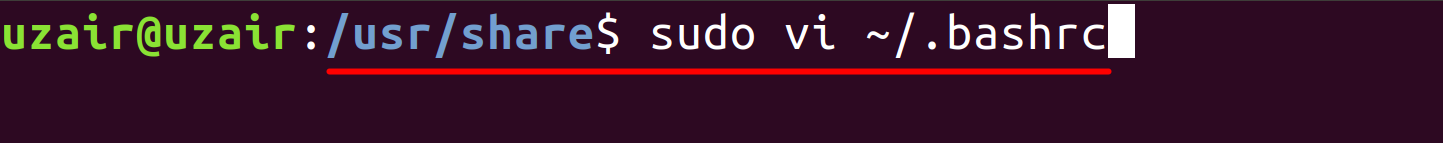
This completes the installation of ZooKeeper. Let us now proceed to next task to configure ZooKeeper.

**Task 2 is complete!**

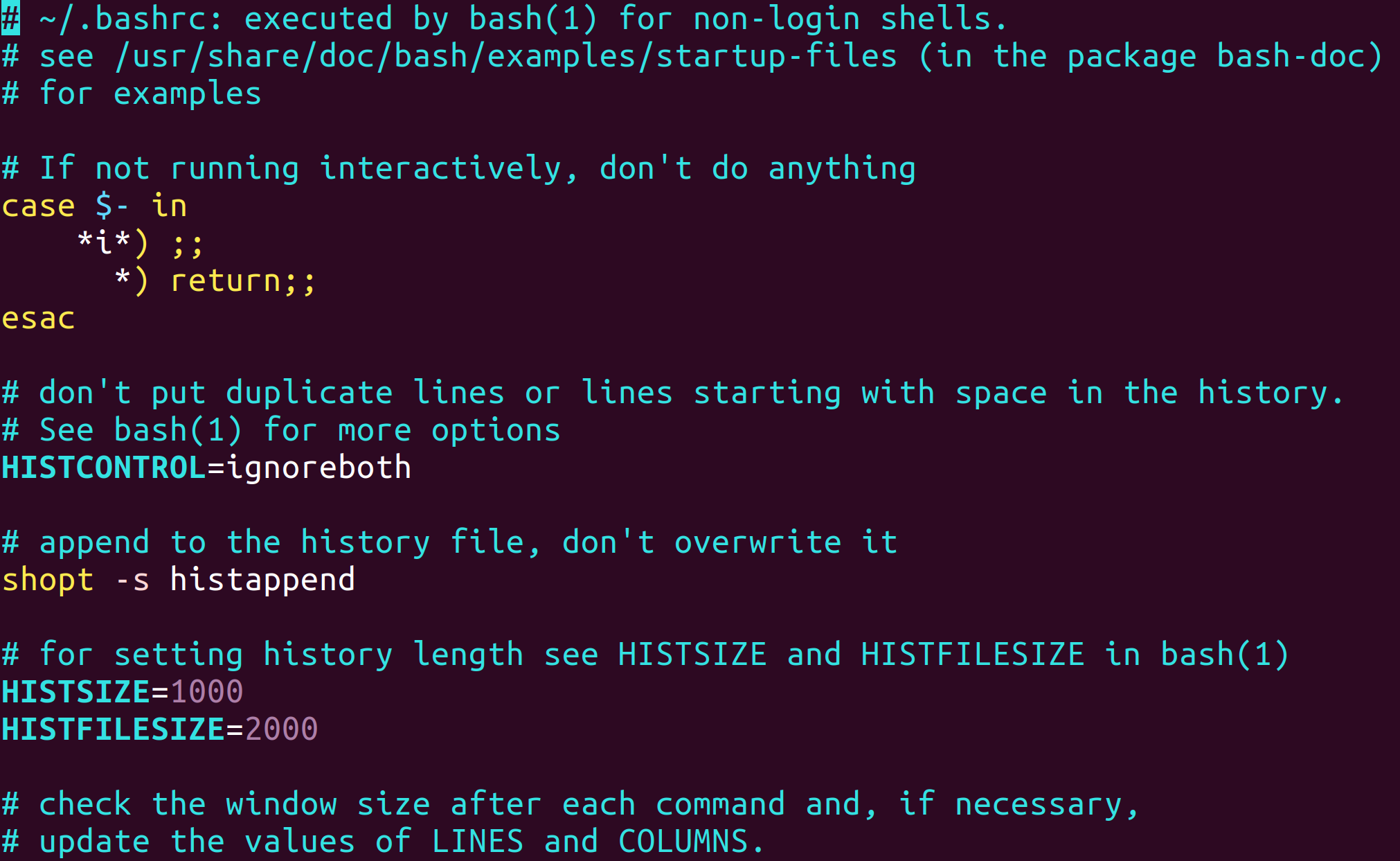
## Task 3: Configure ZooKeeper

**Step 1:**Let us now set up the environment variables for ZooKeeper. Execute the following command to do so.

$ sudo vi ~/.bashrc



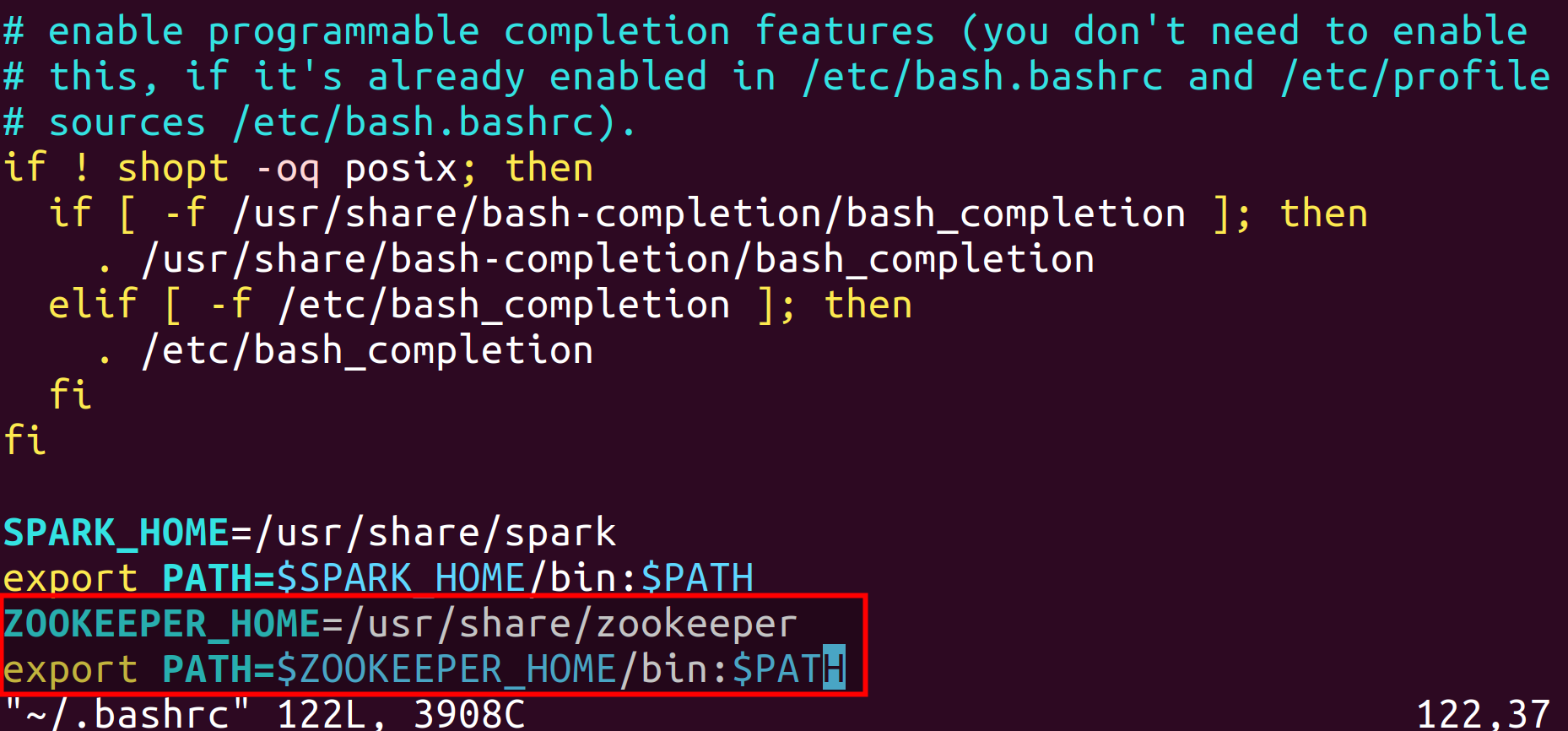
The file should open as shown below.



Now press *i* key to edit the file and append the following environment variable at the end of the file.

ZOOKEEPER\_HOME=/usr/share/zookeeper

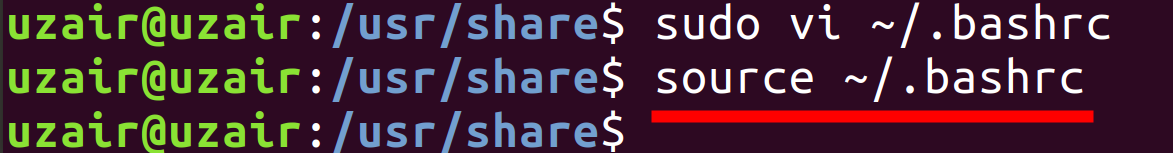
Export PATH=$ZOOKEEPER\_HOME/bin:$PATH



After you have finished appending the text above, hit the *Esc* on your keyboard to stop editing and then press *Shift - Z - Z* to exit out of the editor by saving the changes. (Please see that you need to press Z twice while holding Shift key.)

Now reload the modified *.bashrc* file using the following command.

$ source ~/.bashrc



**Step 5:** Finally, let us create a ZooKeeper configuration file to include the data directory we created in the previous task along with other information. To create the configuration file run the following command from the command line interface.

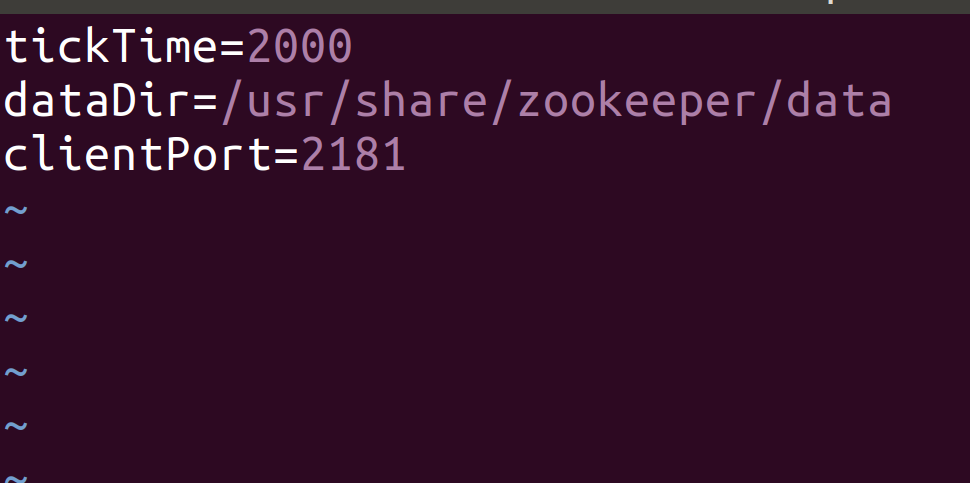
$ sudo vi zookeeper/conf/zoo.cfg

Enter the following settings in the file and save it.

tickTime=2000

dataDir=/usr/share/zookeeper/data

clientPort=2181



The above settings are enough for configuring ZooKeeper in standalone mode.

However, If you wish to install ZooKeeper in replicated mode, please enter the following settings in the configuration file and make sure you have the same settings in all the servers in ZooKeeper Ensemble.

tickTime=2000

dataDir=/usr/local/zookeeper-3.4.6/data

clientPort=2181

initLimit=5

syncLimit=2

server.1=<server\_name>:2888:3888

server.2=<server\_name>:2888:3888

server.3=<server\_name>:2888:3888

.

.

.

**tickTime**: The basic time unit in milliseconds used by Zookeeper. It is used to do heartbeats and the minimum session timeout will be twice the *tickTime*.

**dataDir:** The location to store the in-memory database snapshots and, unless specified otherwise, the transaction log of updates to the database.

**clientPort**: The port to listen for client connections.

**initLimit:** It is the total amount of time allowed forthe quorum members (followers) to connect to and sync with the leader. If most of the quorum members fail to sync with the leader during this period, the leader powers are revoked and a new election for leader takes place.

**syncLimit:** It is the total amount of time allowed for the quorum members to sync with the leader. If the quorum member fails to sync during this period, it will restart itself. Clients will be routed to other quorum members if they were connected to this quorum member.

Next we specify the servers in ensemble each per line with a server number. There are two port settings. First port is used by followers to connect to the leader and the second is used for leader election. There are 3 port numbers to which servers listen on. The description is as follows.

2181: Port for client connections.

2888: Used by followers to connect to the leader.

3888: Used for leader election.

Additional to these settings, you will also have to create a file called *myid* in the data directory which contains a numeric identifier for each ZooKeeper server in the Ensemble. The range for that numeric identifier is 1 to 255. This denotes the server number set in the configuration file above. If you are creating the *myid* file in server.1 just enter 1 in the *myid* file and save it. Similarly, if you are creating the *myid* file in server.2 just enter 2 in the *myid* file and so on.

$ sudo vi zookeeper/data/myid

Enter the numeric identifier ranging from 1 to 255 in the file depending upon the server configuration file.

As soon as we start the server, it reads the *myid* file and determines which server it is based on the numeric identifier. Then it reads the configuration file for all the information regarding the ports and network addresses of other servers within the Ensemble.

**Step 6:** Change the ownership of the files zoo.cfg and myid (replicated mode), by running the following command.

$ sudo chown <username> zookeeper/conf/zookeeper.cfg

$ sudo chown <username> zookeeper/data/myid

***Run the above command only if you are installing ZooKeeper in replicated mode.***

This completes the installation of ZooKeeper.

**Task 3 is complete!**

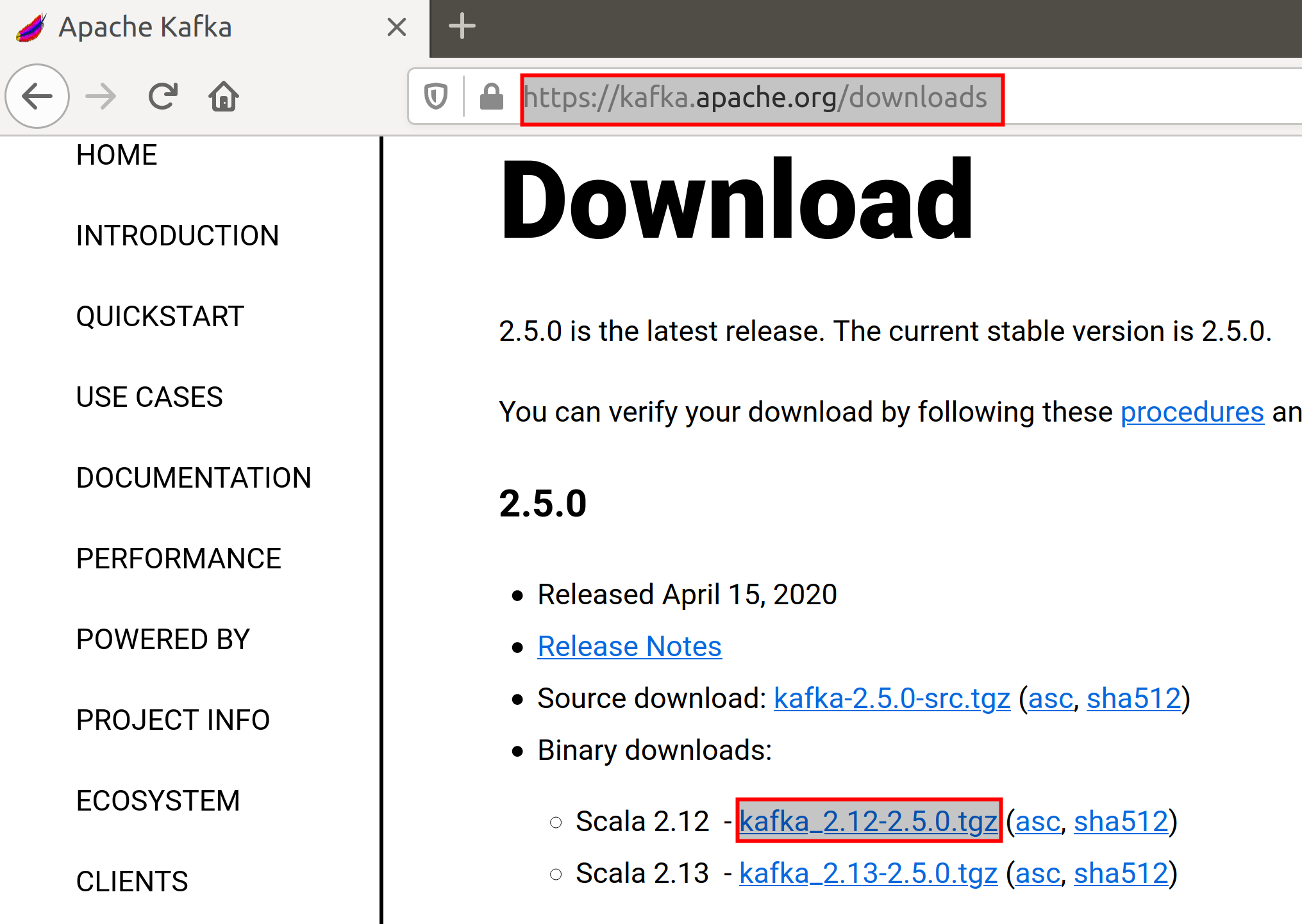
## Task 4: Download and Install Kafka

Now that we have ZooKeeper installed and configured, let us move ahead and install Kafka.

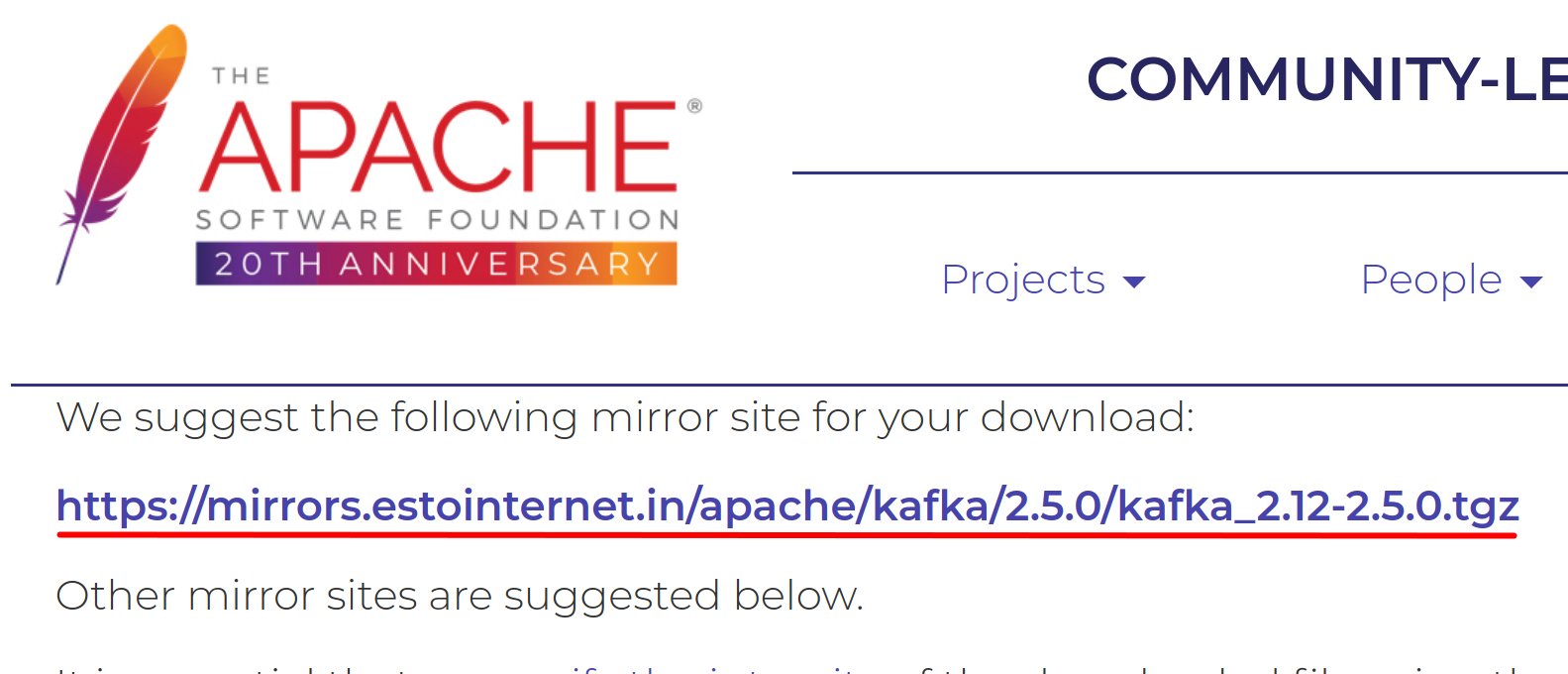
**Step 1:** Let us Kafka in standalone mode. Navigate to the download URL below and click to download the latest stable version for Kafka (which is Kafka 2.12-2.5.0 at the time of writing this book).

Note: 2.12 is the Scala version and 2.5.0 is Kafka version.

Download URL : <https://kafka.apache.org/downloads>



After clicking the download link, you will be taken to a page with mirror site to download Kafka. Click the mirror link as shown below and your download should start. The download may take a while depending upon your internet connection.



**Step 2:** The download will be saved to the *Downloads* directory by default. Execute the following command from your terminal to change the directory to *Downloads* folder.

$ cd

$ cd Downloads

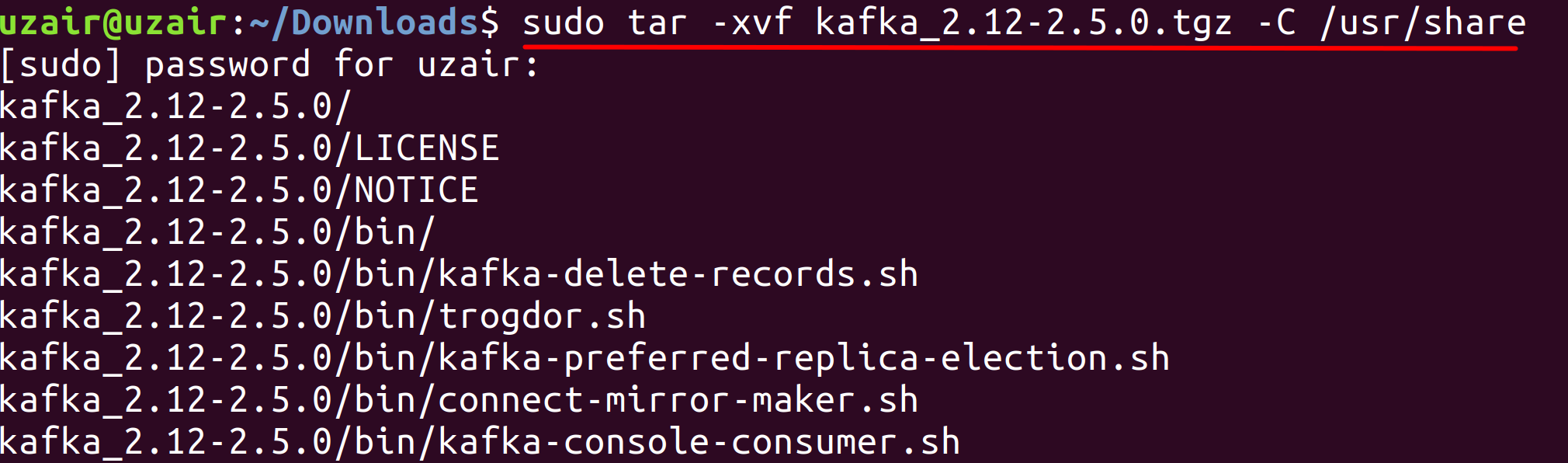
Once you are in the *Downloads* directory, you may optionally check if Kafka has been downloaded using the *ls* command.

$ ls

Now that you are sure that you have the Kafka tar file, untar the Kafka tar file to */usr/share* directory using the command below.

$ sudo tar –xvf apache-zookeeper-3.6.1-bin.tar.gz –C /usr/share

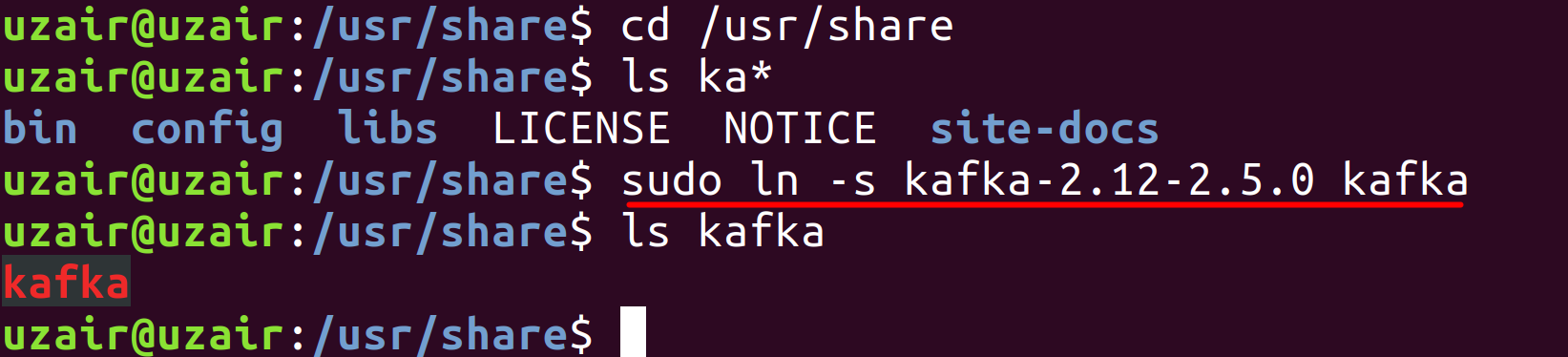
The file will start to untar to */usr/share* directory as shown in the screenshot below.



**Step 3:** Let us make a softlink to the Kafka directory so that we don’t have to refer Kafka with entire name as above. This will also be useful for the future updates. Execute the following command.

$ sudo ln –s kafka\_2.12-2.50 kafka

Run the ls command again to check if we were able to create the softlink successfully.



**Step 4:** Change the permissions of the Kafka directory by running the following command:

$ sudo chown <username> /usr/share/kafka

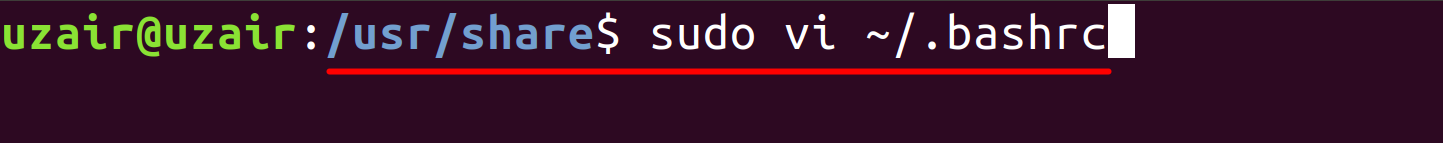
<username> - Insert your username.

**Task 4 is complete!**

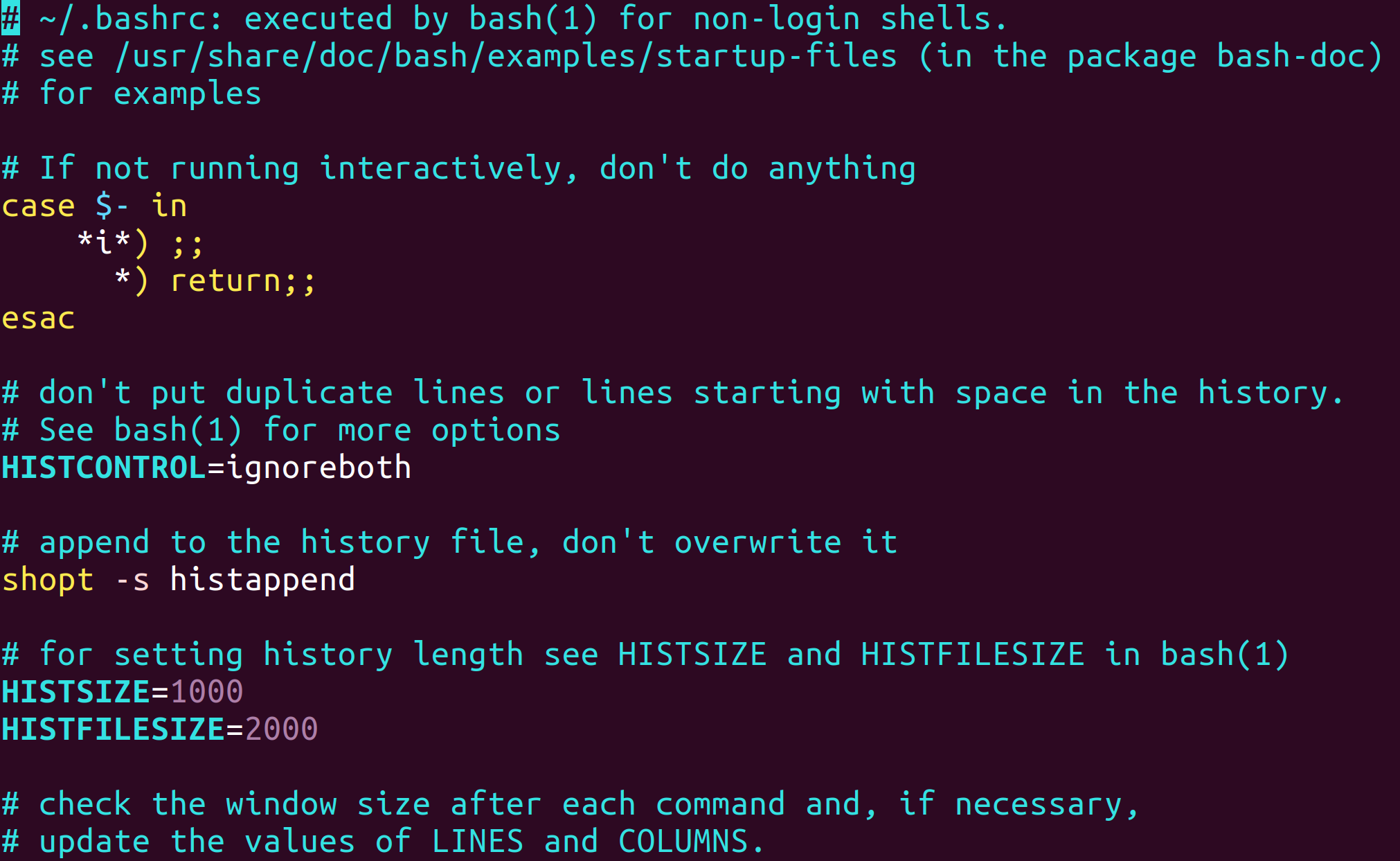
## Task 5: Configure Kafka

**Step 1:**Let us now set up the environment variables for Kafka. Execute the following command to do so.

$ sudo vi ~/.bashrc



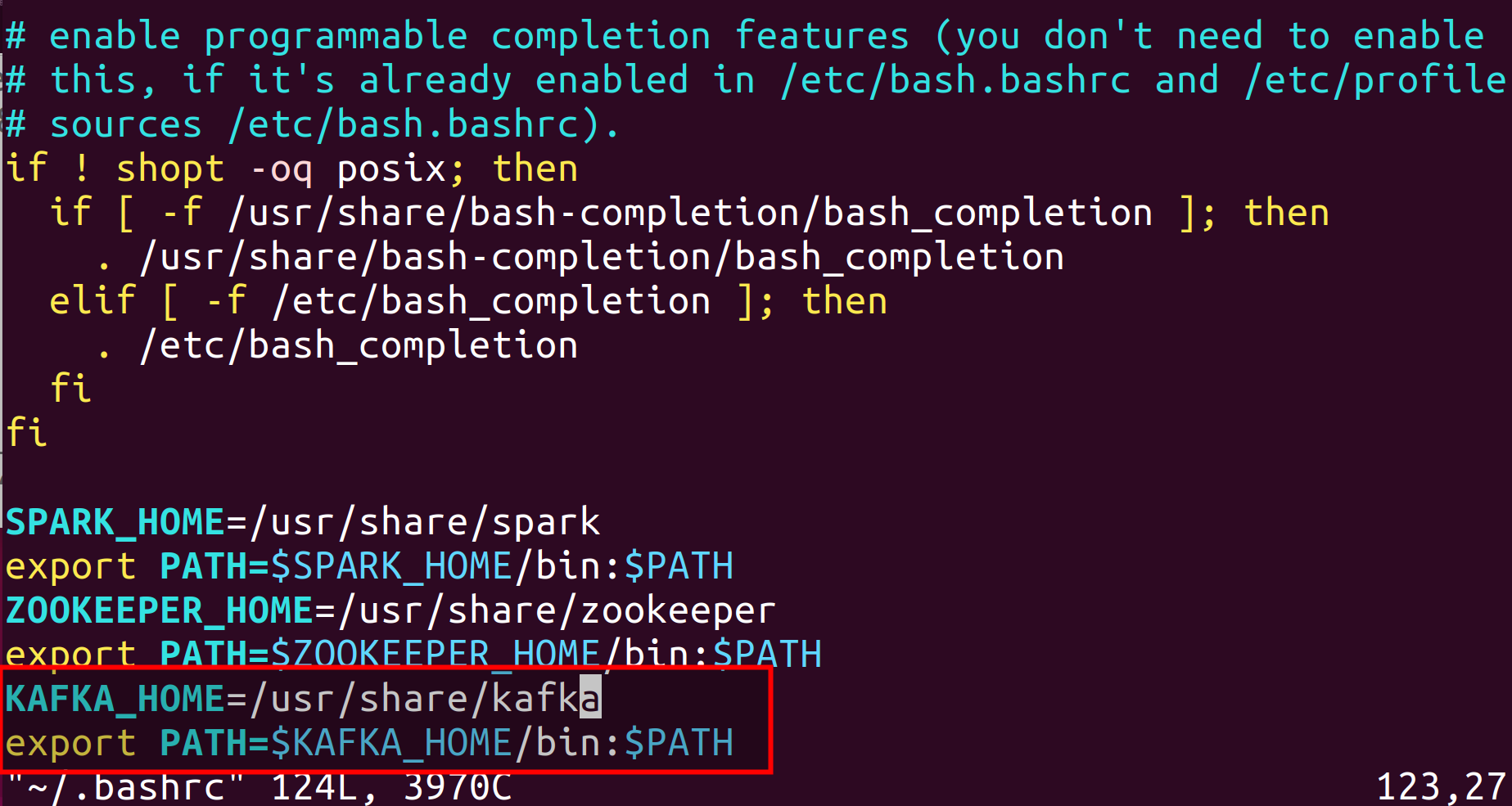
The file should open as shown below.



Now press *i* key to edit the file and append the following environment variable at the end of the file.

KAFKA\_HOME=/usr/share/kafka

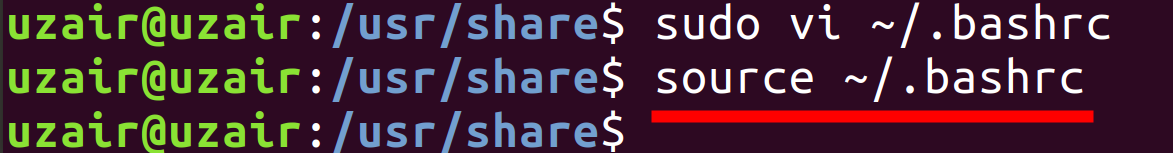
Export PATH=$KAFKA\_HOME/bin:$PATH

****

After you have finished appending the text above, hit the *Esc* on your keyboard to stop editing and then press *Shift - Z - Z* to exit out of the editor by saving the changes. (Please see that you need to press Z twice while holding Shift key.)

Now reload the modified *.bashrc* file using the following command.

$ source ~/.bashrc

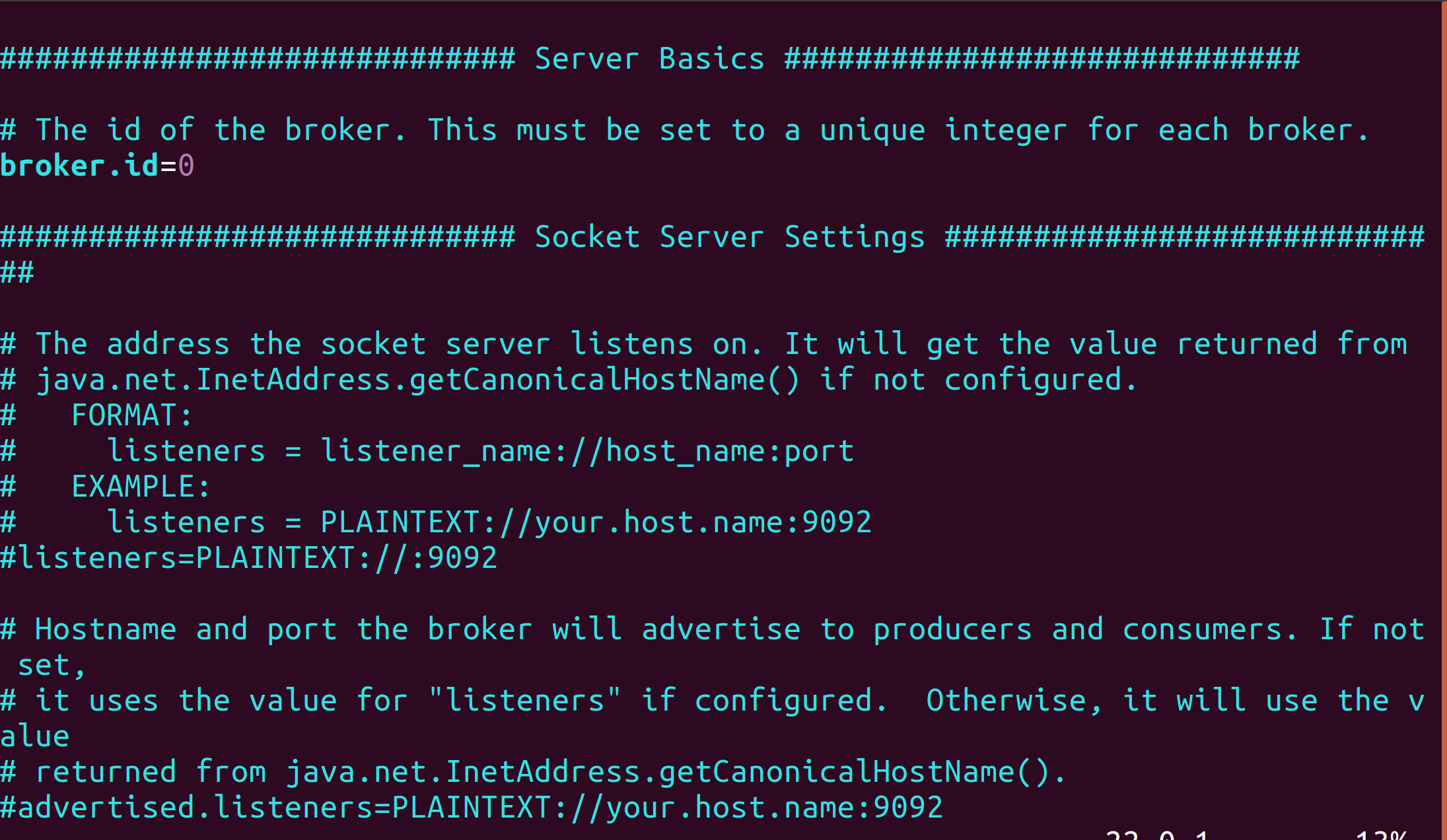


This completes the installation for Kafka in standalone mode. The configuration file provided with the Kafka installation is sufficient to run Kafka for training purposes. However, when working in a real time environment, these configurations are not sufficient.

Let us now look at few of the configuration options available in Kafka.

**Step 2:** The configuration properties for Kafka are available in the following path.

/usr/share/kafka/config/server.properties



**broker.id:** The broker.id is the configuration property that is used to identify a broker in Kafka cluster. The broker.id must be a unique integer value for every single broker in the Kafka cluster and is set to 0, by default.

**listeners:** The listeners property is used to set the URIs as comma separated values, which the brokers will use to create server sockets. The value consists of hostname and port with the default port being 9092. The port number can be changed to any other available port.

**zookeeper.connect:** This configuration property is used to set the connection string of a ZooKeeper server in *hostname:port* format. Multiple hostnames can be provided separated with a comma. The default value is *localhost:2181* for this property. The hostname could be the hostname or ip address of the ZooKeeper server. Optionally, this property can also have a ZooKeeper chroot path as part of its ZooKeeper connection string which puts its data under some path in the global ZooKeeper namespace.

**zookeeper.connection.timeout.ms:** The value set in this property specifies the timeout value for connecting to ZooKeeper in ms. The default value is 18000.

**log.dirs:** We have learnt that messages in Kafka are persistent, i.e., the data received from Producers is stored on disk. This property specifies the location/path for data to be stored. There can be multiple paths specified as comma-separated list. Whenever multiple paths are specified, the broker stores the partitions in the path which has least amount of partitions. The default path is set to */tmp/kafka-logs*.

**num.partitions:** The num.partitions property is used to specify the default number of partitions per topic. The default value for this property is 1.

**num.recovery.threads.per.data.dir:** This property specifies the number of threads that will be used per data directory to recover the data at startup and flushing at shutdown. The default value for this property is 1.

**auto.create.topics.enable**: This property when set to true creates topics automatically whenever applications produce, consume or fetch metadata for a non-existent topic. The automatically created topic will have the default partitions and replications. It is always recommended to set this property to false so that the topics are not automatically created without your knowledge.

**log.retention.hours:** This property specifies the number of hours after which Kafka will delete the messages. There are other similar properties which deal with deletion of logs after a certain period of time. They are *log.retention.minutes* and *log.retention.ms*. All these properties perform the same operation. However, the property having the lowest time unit takes precedence, i.e., *log.retention.ms* has precedence over *log.retention.minutes* and so on. The default value for *log.retention.hours* is 168.

**log.retention.bytes:** This property determines the size of a message upon which it should be deleted. The size after which the log should be deleted has to be specified in bytes.

These are few of the configuration properties for configuring Kafka broker. You can find all the configuration properties in Kafka documentation URL available in the references link.

**Task 5 is complete!**

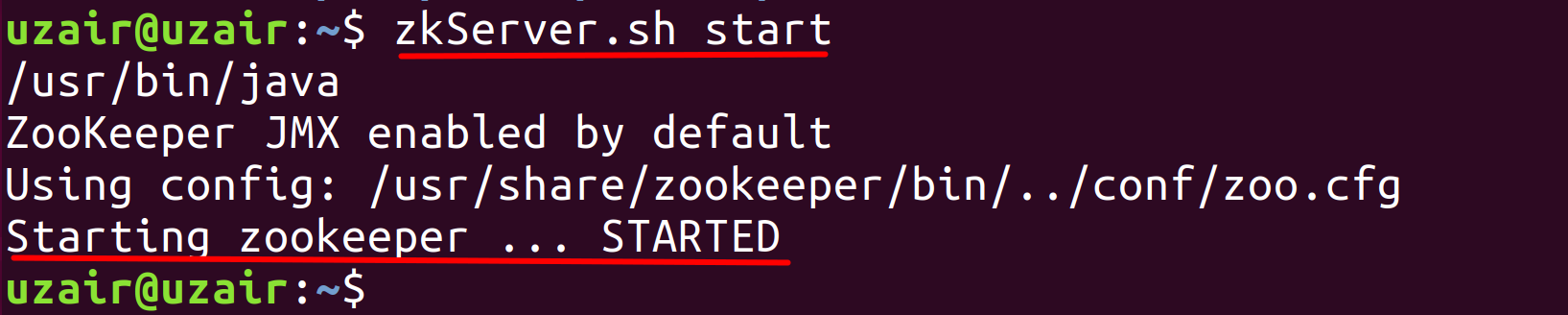
## Task 6: Starting ZooKeeper and Kafka

Now that we have finished installation and configuration of ZooKeeper and Kafka, let us start them.

**Step 1:** First, start ZooKeeper by running the following command.

$ zkServer.sh start

You should see that the ZooKeeper server starts as shown in the screenshot below.



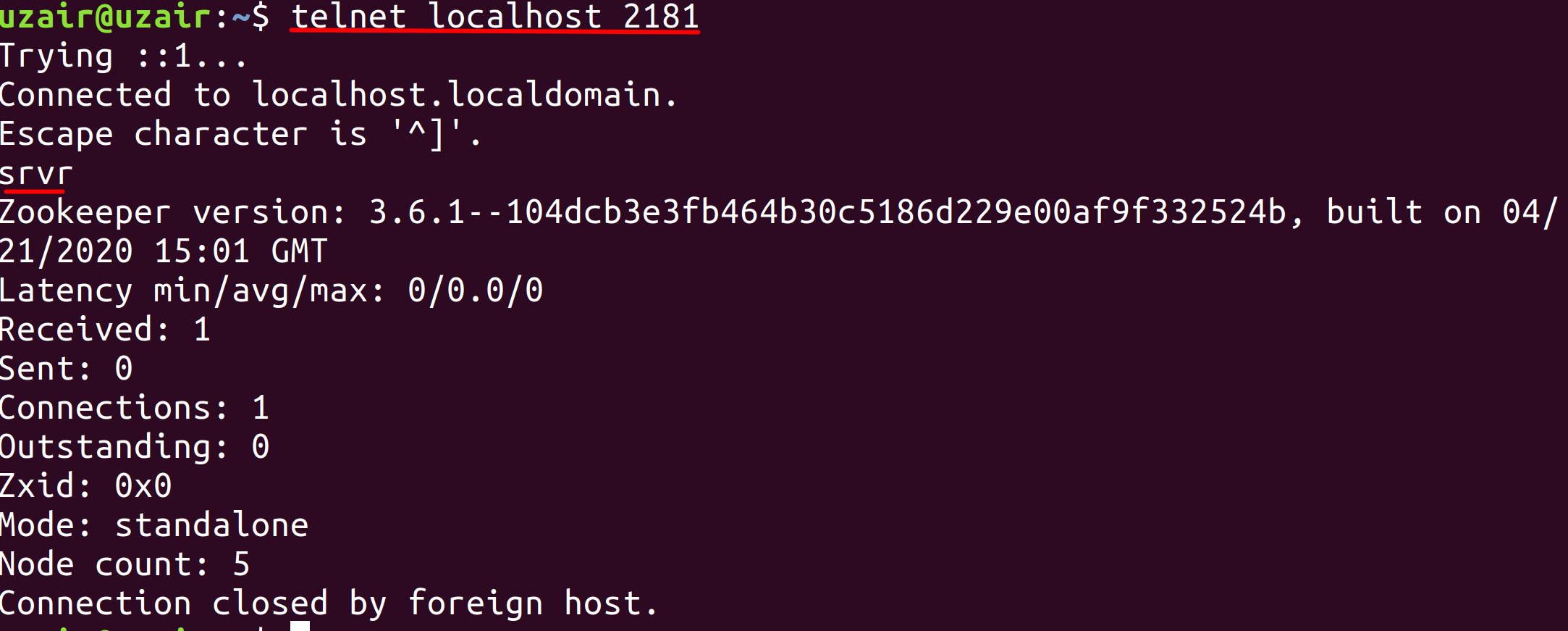
**Step 2:** Let us try connecting to the client port and run the command *srvr*. Run the following command first:

$ telnet localhost 2181

When you see connected to localhost, type the following command:

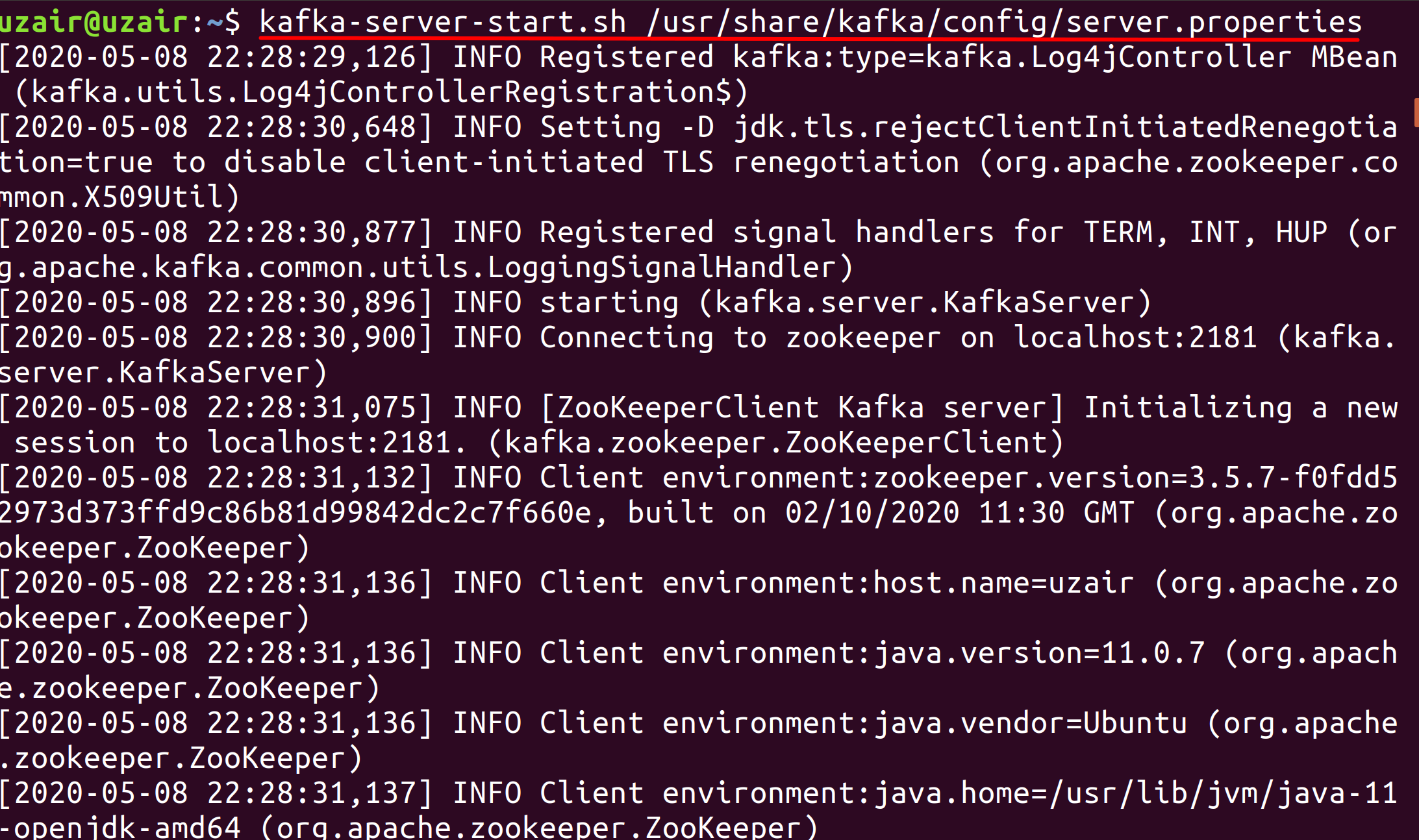
srvr

You should see the output as shown below.

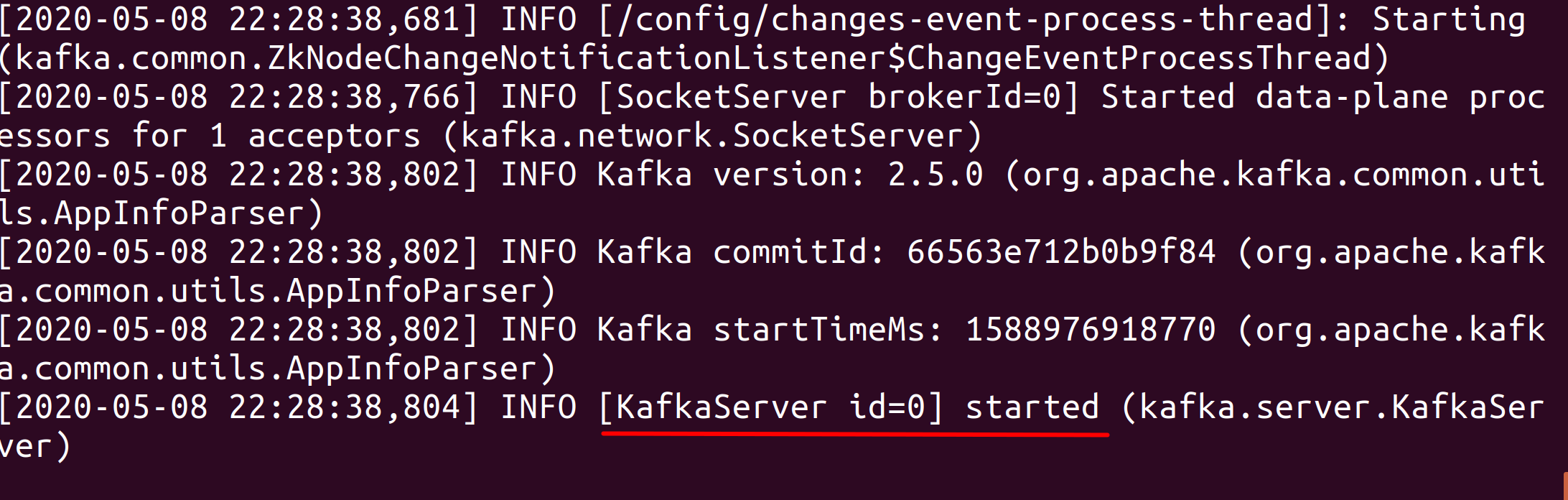


**Step 3:** Let us now start the Kafka server. Run the following command.

$ kafka-server-start.sh /usr/share/kafka/config/server.properties



You should see a lot of information while Kafka starts. The last line would inform you that Kafka has been started as shown below.



**Task 6 is complete!**

# SUMMARY

Kafka is an open-source, distributed, persistent and fault-tolerant message streaming platform or a central repository, which can handle high volume (trillions) of Publish-Subscribe messages every day. Publish-Subscribe messaging system is a system where data is produced (Publish) by producers and consumers consume (subscribe) the data.

The Kafka architecture consists of the following components.

* Brokers
* Producers
* Consumers
* Topics
* Partitions
* Replications
* ZooKeeper

# REFERENCES

* http://kafka.apache.org/
* https://zookeeper.apache.org/
* http://spark.apache.org/
* http://hadoop.apache.org/
* https://www.confluent.io/
* https://kafka.apache.org/downloads
* https://kafka.apache.org/documentation/#configuration